Fall 1-31-1997

# Abstraction of an object-oriented vocabulary by providing a standardized interface

Hemant Kothavade
*New Jersey Institute of Technology*

# Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

# ABSTRACT

## ABSTRACTION OF AN OBJECT-ORIENTED VOCABULARY BY PROVIDING A STANDARDIZED INTERFACE

by
**Hemant Kothavade**

Controlled vocabularies are ubiquitous in varied application fields. They are particularly helpful in the medical field since they can unify disparate terminologies and provide information in a compact, comprehensible manner. In this thesis, we present a mechanism to efficiently retrieve and update knowledge stored in a controlled vocabulary modeled as an Object-Oriented Database (OODB) system. We aim to provide a standardized interface to the vocabulary, such that the implementation details of the vocabulary are transparent to all users. The user of this standardized interface will typically be an application programmer who is trying to provide the vocabulary's knowledge-base to end users. We first describe our approach to creating the standardized interface. We then present the software architecture and design for it. We conclude by describing the implementation of this standardized interface.

# ABSTRACTION OF AN OBJECT-ORIENTED VOCABULARY BY PROVIDING A STANDARDIZED INTERFACE

by
Hemant Kothavade

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer and Information Science

January 1997

APPROVAL PAGE

# ABSTRACTION OF AN OBJECT-ORIENTED VOCABULARY BY PROVIDING A STANDARDIZED INTERFACE

Hemant Kothavade

---

Dr. Yehoshua Perl, Thesis Advisor — Date
Full Professor of Computer and Information Science, NJIT

---

Dr. James Geller, Thesis Co-advisor — Date
Director of Artificial Intelligence and OODB Laboratory
Associate Professor of Computer and Information Science, NJIT

---

Dr. Michael Halper, Committee Member — Date
Assistant Professor of Math and Computer Science,
Kean College of New Jersey

# BIOGRAPHICAL SKETCH

**Author:** Hemant Kothavade

**Degree:** Master of Science in Computer Science

**Date:** January 1997

**Undergraduate and Graduate Education:**

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1997
- Bachelor of Science in Computer Engineering,
  Ramrao Adik Institute of Technology, Bombay, India, 1993

**Major:** Computer Science

This work is dedicated to
my family and friends

## ACKNOWLEDGMENT

I would like to thank Dr. Y. Perl, Dr. J. Geller and Dr. M. Halper for this opportunity to conduct research under their able guidance. Their continuous interest and encouragement have contributed significantly to the work presented in this thesis. It has been an enriching experience for me.

I would also like to thank all my colleagues in the laboratory. A special thanks to the members of my group; without their hard work and dedication this work would not have been possible.

# TABLE OF CONTENTS

Chapter Page

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

A controlled vocabulary is a centralized software system for representing the concepts used in an application domain. Controlled vocabularies are very useful in domains where many different terminologies coexist. The medical field represents such a domain. The increasing use of controlled vocabularies in the medical field will necessitate their efficient implementation and relatively simplified access to the knowledge stored in the vocabulary.

We have implemented a medical vocabulary as an Object-Oriented Database (OODB). This vocabulary originally existed as a semantic network. The mapping from the semantic network to a schema for the OODB is described in Chapter 2.

This thesis focuses on the issues involved in retrieving information from the vocabulary implemented as an OODB. In Chapter 3, various alternatives for information retrieval are considered. As a first step, we identify the possible users of the vocabulary and their specific requirements. Based on this knowledge of user requirements and the vocabulary's modeling, we decide on a specific approach. This approach requires the creation of a set of low level access functions. These functions can be invoked by the programs that supply a specific user interface. Each low level function represents a basic query posed by the vocabulary user. The derivation of the set of all queries and their mapping to corresponding functions is also described.

Chapter 4 describes the design of the set of low level functions. Additional classification of the functions - schema level and instance level - is introduced. Design issues like function names, function argument(s) and error handling are also discussed. A mechanism to share the functions amongst the user interface programs is described.

The implementation issues for these low level functions are discussed in Chapter 5. Accessing the vocabulary for information retrieval, adding auxiliary information and making all the functions shareable are some of the implementation issues discussed. Special measures to make the implementation robust and efficient are also described.

Future work and concluding remarks are available in Chapter 6.

Appendix A provides implementation details like special compilation requirements and accessing the right header files.

Appendix B contains the source code for the implementation of the argument classes and some low level functions.

# CHAPTER 2

# THE OOHVR

## 2.1 Controlled Vocabulary as an OODB

The following material is derived from [20].

Starting from the first generation of semantic networks [4, 26, 28] and semantic data models [12, 13], attempts have been made to computerize the semantics of natural language terms. While most of these attempts were limited to small domains or "toy" applications, there have been a number of notable exceptions such as Cyc [19] and WordNet [21]. Another large semantics-based vocabulary called the Medical Entities Dictionary (MED) has been developed in the healthcare arena [5]. From an application standpoint, controlled vocabularies alleviate software systems of the burden of maintaining their own *ad hoc* vocabularies. A common, centralized vocabulary also facilitates communication among applications by eliminating costly and time-consuming translation tasks. From a user point of view, they can help standardize information processing among different organizations and thus reduce the overall cost of doing business [20].

We have modeled and implemented a semantic network-based controlled vocabulary as an Object-Oriented Database (OODB) [16, 29]. We have chosen to focus on an existing medical vocabulary called the InterMED, an offshoot of the MED [5]. One reason for our choice is the fact that the healthcare field is one where such vocabularies are becoming ubiquitous and are being exploited in a wide variety of settings. We refer to the OODB obtained by this mapping as the Object-Oriented Healthcare Vocabulary Repository (OOHVR). At present, a version of the OOHVR is up and running as an ONTOS [25] database [20].

There are a number of reasons why one would want to model a vocabulary in an OODB. First, in applications where external agents such as intelligent information-locators, decision-support systems, and end-user browsers are demanding the

knowledge stored in the vocabulary, transparent and concurrent access to it is necessary. OODB systems provide the traditional access support of database systems and offer a "low impedance" pathway [29] to the network, particularly at a time when more and more application programs are being built using object-oriented programming languages. The vocabulary can also be accessed declaratively using an SQL extension (like OSQL of ONTOS [24]) or a "path" language such as XQL [15]. Additionally, from a theoretical standpoint, the typical OODB system's repertoire of modeling constructs neatly captures the modeling features of semantic networks. Thus, the vocabulary can be mapped almost directly from the semantic network into the OODB system without having to re-model it from scratch [20].

An additional benefit of using the OODB framework turns out to be increased comprehension of the overall hierarchy and connectivity of the InterMED, which currently comprises about 3,000 terms. Eventually, the InterMED might be expanded to include much of the content of the MED which contains 46,000 concepts. We have previously availed ourselves of such comprehension for identifying and correcting inconsistencies and errors in the MED [10] and to restructure and refine the MED [20].

Previously, an object-oriented framework has been used as a modeling vehicle for thesauri for (natural) language-to-language translation [7, 8]. A terminology editor called TEDI was built in the same context as a tool for extracting relevant information from hypermedia documents [22]. The $O_2$ OODB system has been used to store portions of a general English dictionary based on a "feature structure" description of its entries [14, 20].

In addition to the InterMED, the medical field has seen a number of standardized vocabularies such as SNOMED [6], ICD9-CM [27], and MeSH [23]. A descriptive semantic network called Structured Meta Knowledge (SMK), employing a termino-

logical knowledge-base, has been used to capture the semantics of patients' medical records [9, 20].

## 2.2 Description of the InterMED

In this section, we describe the InterMED, a controlled medical vocabulary modeled as a semantic network. The InterMED is the successor to the MED, which was developed and is presently in use at Columbia-Presbyterian Medical Center. It is being built as an inter-organizational vocabulary to be employed by various medical centers. Structurally, the InterMED is a semantic network whose nodes are medical concepts. Each node can have properties which are referred to as either attributes or relationships. An attribute is a property whose value is a primitive data type (such as a string). A relationship has as its value a reference to another concept in the network. One attribute common to all nodes is *name*, which holds a concept's associated *term* (or textual denotation). Another is *synonyms* which can hold alternate denotations aside from the primary one [20].

The InterMED features a concept subsumption hierarchy—a directed acyclic graph (DAG) composed of concepts connected through super-concept (and sub-concept) links. This hierarchy acts as the property inheritance mechanism within the network. A sub-concept inherits all the properties of it's superconcepts. For example, **Glucose Test** is a subconcept of (or, simply, "IS-A") **Test**, and therefore it inherits all of **Test**'s properties. In other words, the set of properties of **Glucose Test** is a superset of the properties of **Test**. A concept may have more than one parent concept. Also, the entire vocabulary hierarchy is rooted at a single concept called **Entity** [20].

The second purpose of the hierarchy is to support reasoning. Such a capability would be exploited, for example, by decision support systems that make subsumption-based inferences [20].

At present, the InterMED comprises about 3,000 medical concepts. This figure is expected to increase into tens of thousands as the InterMED is extended over time to cover much of the current content of the MED. The concepts are linked by approximately 9,000 non-hierarchical (i.e., non-IS-A) relationships. The IS-A links total about 5,000 [20].

## 2.3 Initial OOHVR Schema

The concepts in the semantic network have been assembled into structural groups. Each group contains all the concepts that share the same properties. In the context of the OODB, each group can be defined as a class and all the concepts for that group will be instances of that class. Thus, some nodes in the semantic network serve as the basis for the definition of object classes in the OOHVR schema, while all nodes are mapped directly into instances of those classes [20].

The question is: Which nodes of the InterMED will actually guide the definition of classes and their associated properties? Because the purpose of an object class is, among other things, to define the properties of it's instances, it is sensible to examine the nodes of the network that also function in this role. Ultimately, our principal task is to identify groups of nodes that share identical properties so that we can define the OOHVR schema's classes [20].

It turns out that there are only 30 concepts of the InterMED that introduce properties. We will call these *property-introduction nodes*. The rest just inherit their properties from other concepts. Because only 30 out of the nearly 3,000 nodes introduce new properties, we call the InterMED's subsumption hierarchy a *sparse inheritance hierarchy*. Vocabularies, in general, by their very nature tend to have sparse inheritance hierarchies. This situation is in sharp contrast to the subclass hierarchy of a typical OODB schema where at almost every class we expect to find the definition of new properties [20].

We define an area to be a property-introduction node and all its descendants down to but excluding its direct-property-introduction descendants. The property-introduction node of an area will be called its root, and will be used to denote the area.

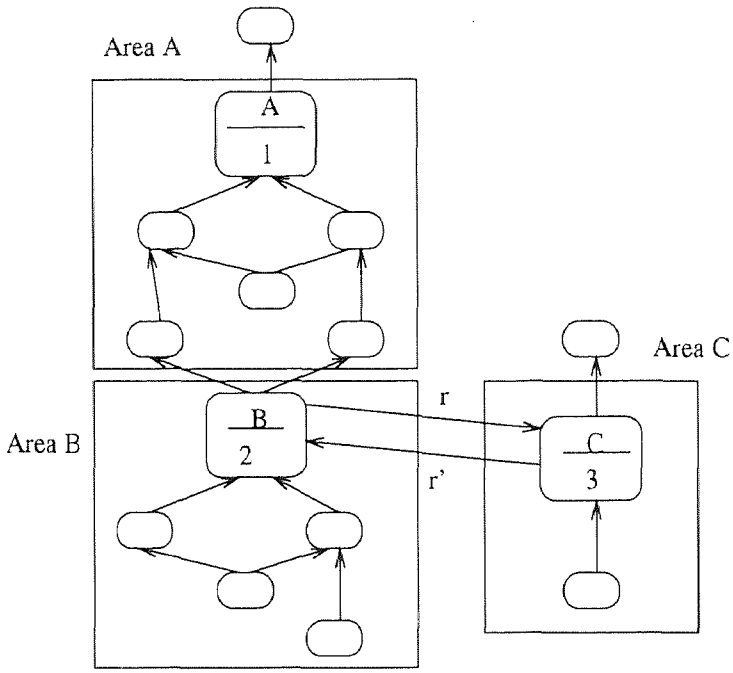


**Figure 2.1** Three areas in a vocabulary

In Figure 2.1, we show three areas $A$, $B$, and $C$ of a vocabulary. The nodes are represented as small rectangles with rounded edges, while the areas appear as large rectangles enclosing their respective nodes. Note that the root of area $A$ (i.e., the node **A**) introduces the single attribute "1," listed inside its rectangle. Area $A$ extends down to, but excludes, node **B** which is a direct-property-introduction descendant of **A**. **B** defines the attribute "2" as well as the relationship $r$ (drawn as a labeled arrow) and serves as the root of area $B$. Finally, area $C$ has the root **C** which introduces attribute "3" and relationship $r'$, the converse of $r$. The IS-A links

are drawn as unlabeled arrows directed from the sub-concept to the super-concept [20].

As a concrete example from the InterMED, the concept **Measurable Substance** introduces a new relationship *measured-by* and is thus the root of a new area. All descendant concepts between **Measurable Substance** and its direct-property-introduction descendants are in this "Measurable Substance" area. Examples of such concepts are **Color**, **Temperature**, **Specific Gravity**, **Viscosity**, **Blood Coagulation**, and **Optical Density** [20].

As we noted above, the InterMED has the concept **Entity** as the root. **Entity** introduces a number of properties, and it therefore is the root of the "Entity" area. We will call this area the *root area* of the vocabulary. Because there are 30 property-introduction nodes in the InterMED, it is divided into 30 areas. With respect to the overall size of the vocabulary—approximately 3,000 nodes—this is a very compact division. If all areas are mutually disjoint (i.e., no concept appears in more than one), then all nodes in an area will have the exact same properties (specifically those defined or inherited by its root), and areas will provide the partition we need to define the classes of the OOHVR. We will, in the remainder of this section, describe the OOHVR schema under the assumption that the areas of the vocabulary form a partition. In the succeeding subsection, we will discuss the additional complexity encountered when areas are not disjoint [20].

Under the assumption that areas are disjoint, we define the OOHVR schema as follows. For each area in the InterMED, we define an object class in the OOHVR whose instances will be exactly the concepts in that area, including its root. The class's intrinsic properties are those defined by the area's root. Because the extension of the class is precisely one area, we refer to it as an *area class*. Therefore, the OOHVR schema comprises area classes. The name of a class is formed by concatenating the name of the area's root concept and "_Area." So, the "Measurable

Substance" area would have the corresponding class *Measurable_Substance_Area*. Its properties would include the relationship *measured-by*, among others [20].

Another issue that needs to be addressed is which area classes should be related via subclass relationships. Because the InterMED is singly rooted, each concept in the InterMED is a descendant of the **Entity** concept. Thus, the root of any area in the InterMED is a child of a node(s) in some other area(s). (The exception being **Entity** itself.) Thus, the root of an area has all the properties of its parents' areas plus the properties that it intrinsically introduces. To capture this in our model, we place each area class corresponding to a root node in a subclass relationship with respect to the area class(es) of its parent(s). It should be noted that because a node (particularly a property-introduction node) may have more than one parent, the subclass hierarchy induced by this process is not necessarily a tree, as it may exhibit multiple inheritance. The class *Entity_Area* corresponding to the "Entity" area appears as the root of the OOHVR schema [20].

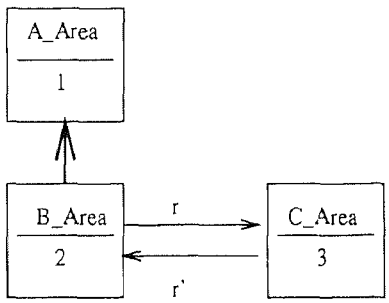


**Figure 2.2** Areas classes corresponding to three areas in Figure 2.1

To illustrate this approach, we first show the result of mapping the three areas of Figure 2.1 into corresponding area classes and subclass relationships in Figure 2.2. Then in Figure 2.3, we show the entire OOHVR schema. Both figures were drawn using our OOdini-2 graphical notation which is based on a schema diagramming

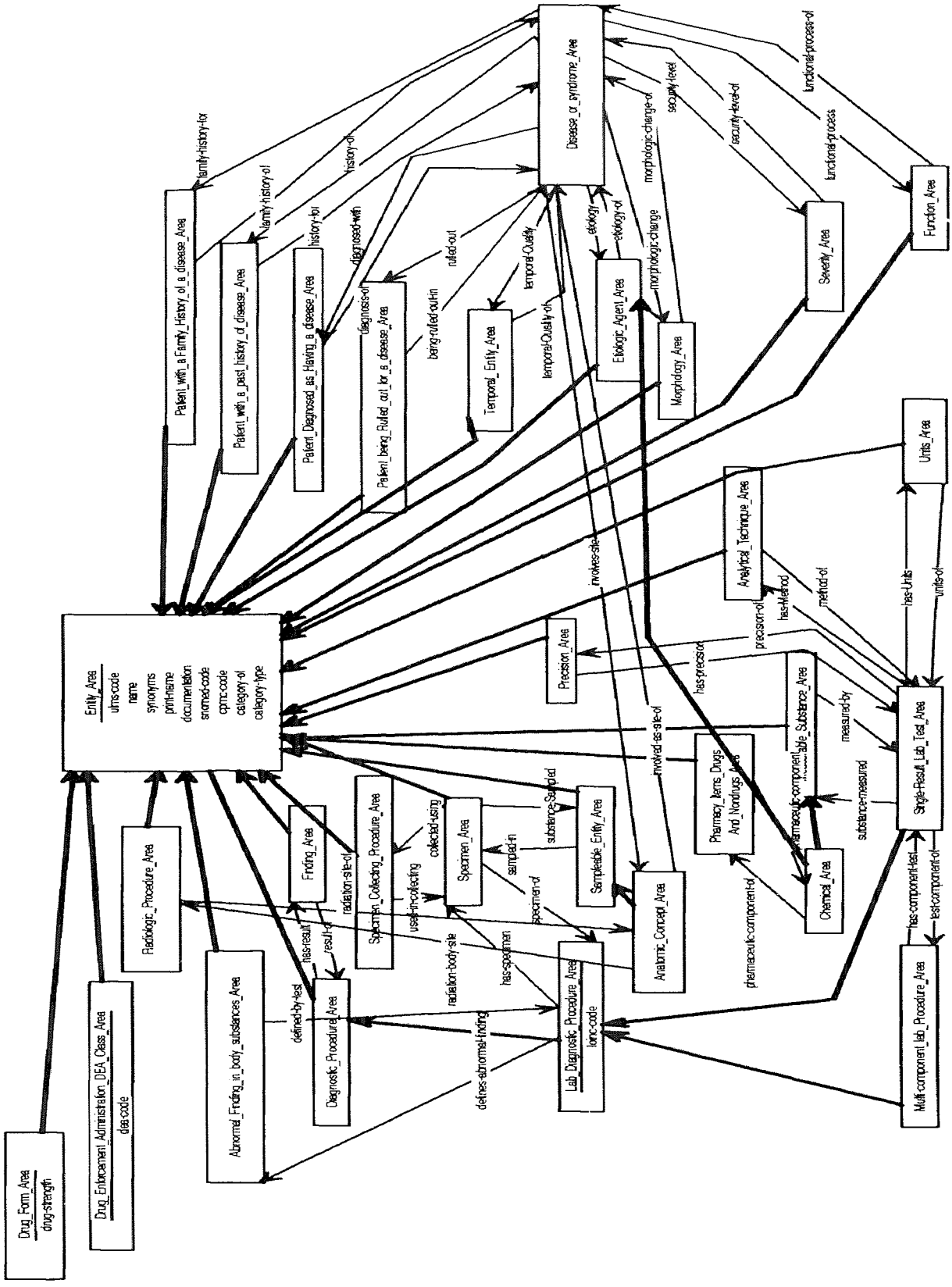


Figure 2.3 OOHVR Schema

language presented previously in [11]. The pictures were produced with the OOdini-2 editor that is being built using the ObjectMaker Tool Development Kit of Mark-V. With OOdini-2, a class is represented as a rectangle, and a relationship, as a labeled thin arrow. We denote a subclass relationship as a bold arrow directed upward from the subclass to its superclass. Attributes are listed inside their respective class rectangles beneath the class name. Let us emphasize again that the OODB schema produced by this mapping turns out to be very compact in terms of the number of classes, particularly when one considers that the InterMED contains thousands of concepts [20].

It is helpful to note that the InterMED's concept subsumption hierarchy served as the basis for the mapping into the OOHVR schema. In fact, the mapping really constituted the identification of the property-introduction nodes and a "collapsing" of the inheritance paths between these concepts. Thus, the OOHVR schema can be seen as an abstraction of the property definitions and accompanying inheritance that occur within the InterMED. For this reason, we call this kind of schema for a sparse inheritance hierarchy a *network abstraction schema* [20].

However, if one is still to use the concept subsumption hierarchy of the vocabulary in the other ways that it was intended (e.g., in order to reason with respect to it), then it is mandatory that it appear in its entirety within the OOHVR. This is accomplished by introducing two reflexive relationships at the root area class *Entity_Area*: *has_superconcepts* and *has_subconcepts*. These properties are defined as follows. In the InterMED, if **X** IS-A **Y**, then, in the OOHVR, the object corresponding to **Y** is a referent of **X** with respect to the *has_superconcepts* relationship; *has_subconcepts* is the converse. In other words, the hierarchy of concepts in the InterMED is represented in the OOHVR on the instance (object) level rather than at the schema level. The schema of the OOHVR provides a compact framework for the definition and inheritance of all the properties of the concepts in the vocabulary. It

thus helps the user of the vocabulary comprehend the vocabulary's overall structure [20].

## 2.4 Extended OOHVR Schema

In the InterMED, some concepts assume membership in more than one area, thus violating the disjointness condition. This multiple membership is due to the fact that each such concept is subsumed by multiple parents (or other ancestors) that reside in different areas. (Recall that this is possible because the vocabulary's concept subsumption hierarchy is a DAG, not a tree.) It should be noted that a node, say, **X** belonging to two (or more) areas cannot be a property-introduction node, otherwise it would be the root of some new area of its own. Instead, **X** would exhibit the combined properties from its multiple areas without introducing any properties that are new [20].

The question is how does this affect the mapping described in the previous section. To see the problem, let us assume that concept **X** resides in the two unrelated areas *A* and *B*. By "unrelated" we mean that neither *A*'s corresponding area class (i.e., *A_Area*) nor *B*'s (*B_Area*) is a descendant of the other in the OOHVR schema. **X**'s dual area membership implies that the object corresponding to it in the OOHVR must be an instance of both *A_Area* and *B_Area*. However, in ONTOS [25], an object cannot be a "direct instance" of more than one class. Thus, we need to modify the mapping slightly in order to accommodate this scenario, which, as it happens, occurs infrequently within the InterMED.

The problem described for **X** is true for any non-property-introduction node whose parents are in different areas or any such node with an ancestor in that situation. In Figure 2.4, we expand the vocabulary pictured in Figure 2.1 to include four nodes that reside in both areas *B* and *C*. Nodes **D** and **E** both have parents in those areas, while **F** and **G** are in the areas by virtue of the fact that they are

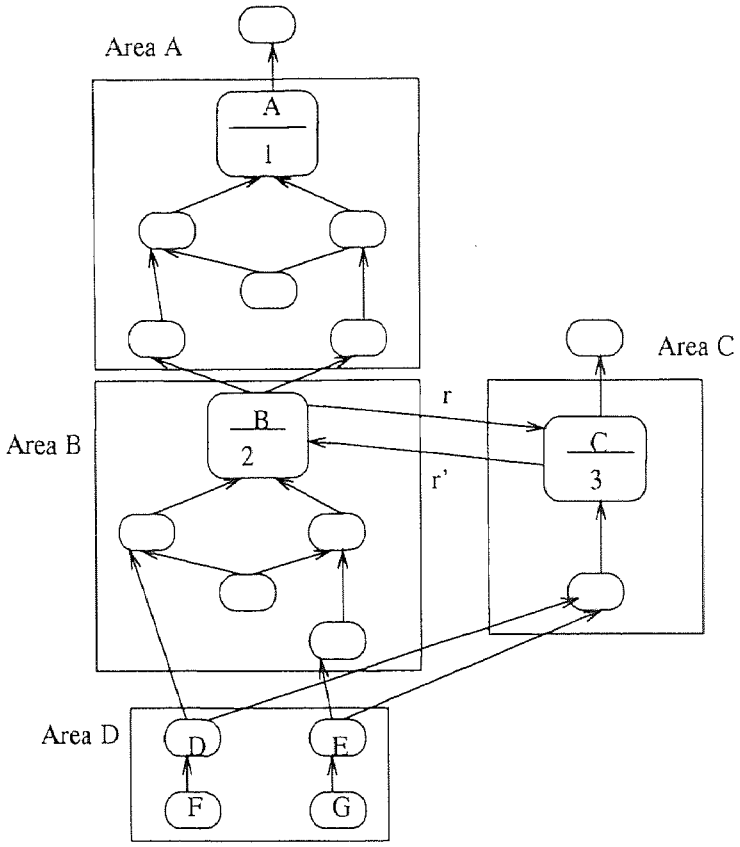


**Figure 2.4** Expanded version of vocabulary from Figure 2.1

children of **D** and **E**, respectively. As a concrete example of this, the InterMED concept **Buffered aspirin tablet preparations** resides in the two areas "Aspirin tablet preparations" and "Drug enforcement administration (DEA) class."

Our solution to the problem is to extend the notion of area and define the non-empty intersection of two areas as an area of its own, called an *intersection area*. As with all other areas in the vocabulary, a class is defined for it in the OOHVR schema. This new kind of class is referred to as an *intersection area class*. The concepts in the intersection area are made instances of this intersection area class, which does *not* introduce any new properties. Instead, the class gets its properties

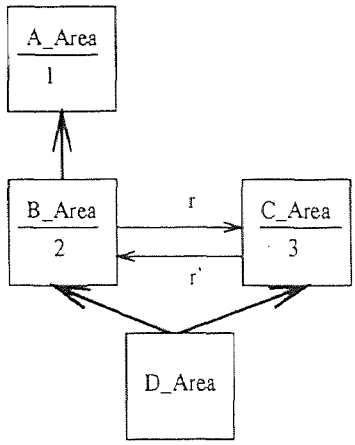


**Figure 2.5** Areas classes corresponding to four areas in Figure 2.4

entirely via inheritance. The required properties are exactly those of the two areas of which it models the intersection. Therefore, the intersection area class is defined as a subclass of the two area classes corresponding to those areas. In Figure 2.5, we illustrate the result of the mapping that the intersection of the areas $B$ and $C$ from Figure 2.4 undergoes in the construction of the OOHVR schema [20].

The notion of intersection area can be extended to encompass the intersection of three or more unrelated areas. In the InterMED, the "Acetaminophen/codeine tablet preparations" area is the intersection of three areas: "Pharmacy items (drugs and nondrugs)," "Drug enforcement administration (DEA) class," and "Drug form." Thus, the intersection area class in this case has three parents in the OOHVR schema. It is also possible for an intersection area class to be a subclass of another intersection area class [20].

As shown in Figure 2.4, the intersection area might not have a root (i.e., a concept which is an ancestor of all others). If there exists a root X, then the corresponding intersection area class will naturally be denoted *X_Area*. Otherwise, the

schema designer will have to select one of the concepts in the intersection as the name of the intersection area class. In Figure 2.5, the name was chosen to be *D_Area*.

In Figure 2.6, we show the entire OOHVR schema, including all intersection area classes. The schema comprises a total of 39 area classes and 50 subclass relationships [20].

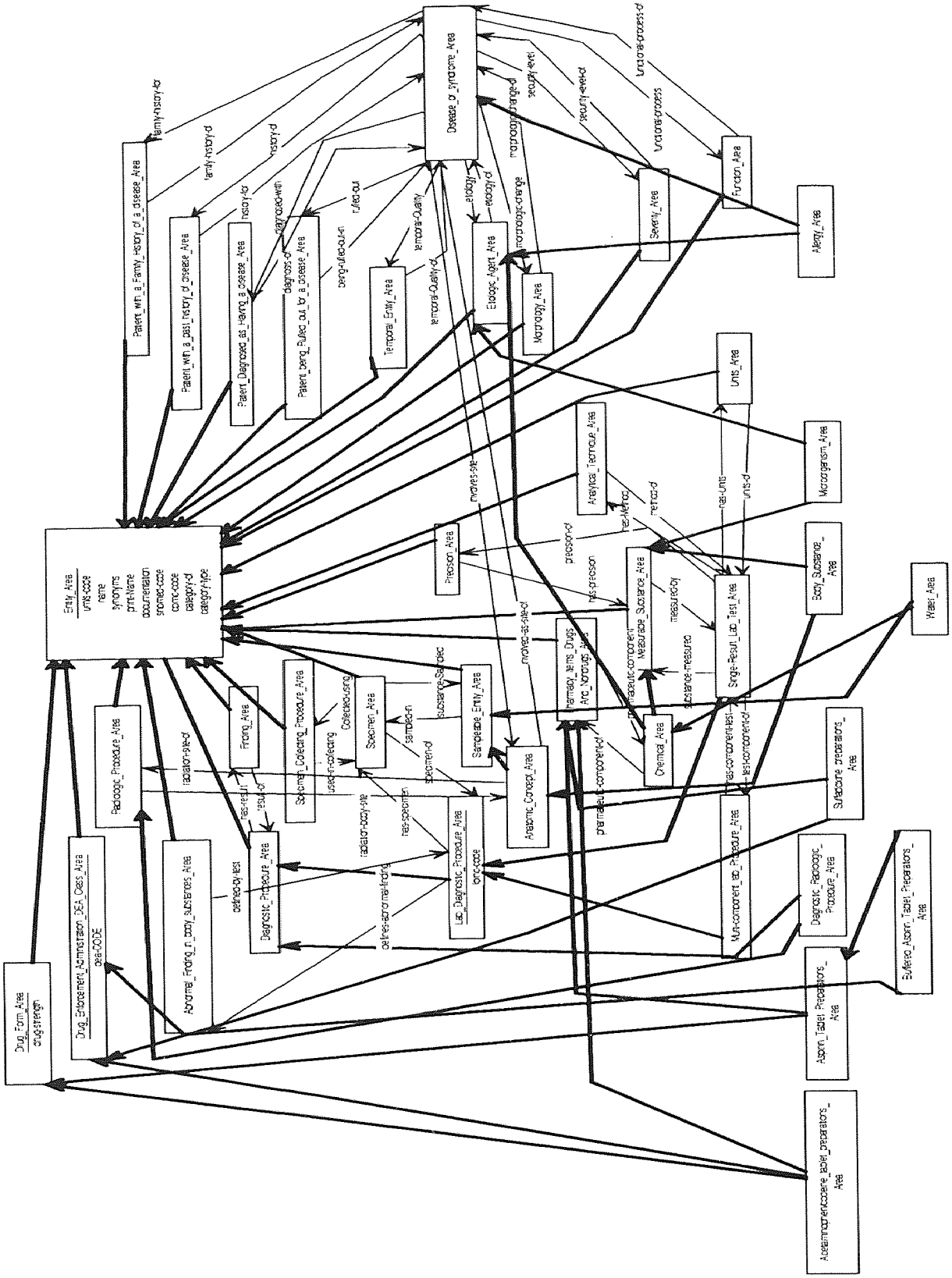


**Figure 2.6** OOHVR schema including intersection area classes

# CHAPTER 3

# AN APPROACH TO THE VOCABULARY INTERFACE

There are many end users for the OOHVR. These users have varied requirements when they access the vocabulary. Hence, multiple user interfaces will have to be provided. Efficient and easy access to the vocabulary is desired by each of these user interfaces.

## 3.1 User Interfaces to the Vocabulary

Presently, we have identified three potential user interfaces to the OOHVR:

- A Form-Based Web Interface
- A Programmatic Interface
- A Graphical Interface

### 3.1.1 The Form-Based Web Interface

The vocabulary is presented as a set of queries. Each query provides information relevant to a particular aspect of the vocabulary [15, 16]. Typically, a query prompts the user to identify the term or area for which information is to be provided. This interface is now available on the World Wide Web.

It should be noted that each query must be capable of handling multiple users simultaneously. For this purpose, this interface requires the code for the queries to be re-entrant. That is, the code for a particular query could have more than one thread passing through it at the same time. We have provided this functionality by using the UNIX *"archive"* facility (section 5.5). For the web programmer, designing and implementing the form based interface is simpler if the information in the vocabulary is available in the form of preliminary data types like a string of characters or an

integer. The web programmer is also concerned with implementation issues like compatibility with the compiler required for accessing the vocabulary.

### 3.1.2 The Programmatic Interface

This interface will typically be used by people with considerable knowledge of the vocabulary. Programs could be written to perform tasks involving a varying degree of complexity [5]. These programs could be tools used by the vocabulary administrator. Alternately, these programs may present some specific knowledge in the vocabulary. The programmer for this interface will typically need to access the vocabulary more than once in order to obtain the information required by such programs.

### 3.1.3 The Graphical Interface

The vocabulary's structure is presented graphically to emphasize its hierarchical object-oriented modeling. This enhances the user's comprehension and facilitates maintenance of the vocabulary. The icons needed for this graphical representation are available in the OODINI [11, 22] software. This interface will need details about the schema of the OOHVR database and the structural and semantic relationships in the database.

## 3.2 Interaction between the Vocabulary and User Interfaces

We see that the vocabulary as an ONTOS database is on one side and the user interfaces are on the other side. We need to provide a mechanism for interaction between these two components. This mechanism can be an *Application Programmer's Interface (API)*. This API provides a list of function headers. For each query posed to the user interface, the appropriate function(s) in the API are invoked and it is the API function's responsibility to retrieve the relevant information from the database [16].

### 3.2.1 Advantages of the API Approach

1. The use of the ONTOS database is transparent to all programmers accessing the vocabulary.

2. All the user interfaces can use the same set of API functions. The API also addresses the specific requirements of each user interface.

3. The whole OODB system becomes "an abstract data type" which makes it easier to update the version of ONTOS or to change the representation of the vocabulary.

## 3.3 Basic Approach to API Development

### 3.3.1 Mapping Queries to Functions

In developing this API, we have two extreme alternatives available. On one extreme, there is a single function that accepts a text string argument. The text string is then parsed and interpreted as a query which is passed on to the appropriate functions that access the ONTOS database. On the other extreme, there is an exhaustive list of all possible queries, and for every query there is a separate function that directly, and relatively easily, accesses the database [15].

Advantages of the single function approach:

- It's easier to add new queries.

- The user interface programs will always know exactly which function should be called.

Disadvantage of the single function approach:

- Parsing the input text string is a complicated problem and is comparable to writing an SQL interpreter with query optimizer and query processor.

Advantage of the multiple functions approach:

- The complex parsing and interpretation problem is bypassed.

Disadvantage of the multiple functions approach:

- Adding new queries is relatively difficult.

The intermediate solution is based on the principle of avoiding oversimplification by having one function per query. We supply a set of functions which permit the expression of most queries by a single API function call or the simple composition of a few API function calls.

### 3.3.2 Abstraction of the Database

We could treat the whole ONTOS implementation of the vocabulary as one very large *abstract data type (ADT)*. As is well known, in an abstract data type a complicated data structure is encapsulated by giving the user no direct access to the data. Indeed, the organization of the data is transparent to the user. The user (the user interface programmer in our case) can access the data only through a relatively small set of well defined access functions. All querying, testing and updating of the data structure (our database) can only be done through such access functions. The designer of the abstract data type has to ensure that he or she supplies all access functions that might be necessary during the lifetime of the ADT. If a user requests any additional access functions, the ADT designer must supply them. On the other hand, as is well known, the internal organization of the data structure (in our case, database) may be changed without altering any of the programs that rely on the data structure. Only the access functions themselves need to be changed. These advantages of abstract data types lead us to rely on a metaphor of the ONTOS DB as an abstract data type with a list of low level access functions [3].

## 3.4 Deriving the List of Functions

### 3.4.1 The Graph-Theoretic Approach to Queries

The *InterMED* [5], a medical entities dictionary, has been used as the source of our vocabulary. Since the InterMED is based on a semantic network, queries are constructed based on the following ideas:

- Path Languages
- Pattern Matching

**3.4.1.1 Path Languages** A path language describes a path with a prototype that might or might not contain variables. Some semantic networks as well as some Object-Oriented systems supply path languages. Once a path has been specified, an interpreter retrieves all or one of the paths conforming to the path description. For an example of a semantic network with a path language, refer to [17, 18].

**3.4.1.2 Pattern Matching** In many Artificial Intelligence systems, especially PROLOG-like systems, expressions with constants and variables can be formulated. These expressions can then be matched against a knowledge base. If there are any variables in the expression, and if that match was successful, the result will consist of one or more copies of the original expression, with variables replaced by values found in the knowledge base [26].

**3.4.1.3 Using Paths and Pattern Matching** Fundamentally, any "large" structure in a semantic network can be described by composing triples of the form (NODE, EDGE, NODE). Smaller structures can be described as NODE or as EDGE, or a pair of the two. Therefore, we need to generate an exhaustive list of such paths, under the assumption that each one is a pattern.

For each path in the list, replace every possible combination of elements either by a literal or by a variable.

For example, NODE - EDGE - NODE can become:

NODE(VAR) - EDGE(CONST) - NODE(CONST) or

NODE(CONST) - EDGE(VAR) - NODE(CONST)

The combinatorial patterns generated correspond to the possible queries posed by the user. These queries will have to be translated into a set of access functions to be included in the API.

### 3.4.2 Filtering the Patterns

The list of access functions is constrained in two ways. First, we need to know exactly what queries will be posed by the various user interfaces. This is the necessity constraint. Second, the functions specified should be implementable. The 'non implementable' functions do not refer to problems that are in principle undecidable or intractable, or for which algorithms are not known. Instead, they refer to problems for which the actual function implementation would require person years of time, or where the writing of the function would require a complete redesign of the ONTOS database, or where the runtime of an implementation would be measured in minutes. These constraints defined by implementation time and run time are the possibility constraints.

Between the necessity constraints and the possibility constraints, we have to find a set of compromises. Some desirable queries may have to be eliminated if they have long run times. Some aspects of the ONTOS database design may have to be extended to accommodate absolutely necessary queries.

### 3.4.3 The Patterns for OOHVR

The combinatorial patterns for the OOHVR can be obtained by applying the techniques discussed above to the InterMED. We treat each node as a class and

each arc as a property (relationship or attribute). For the InterMED, the semantic network triples translate into two generic patterns:

CLASS - RELATIONSHIP - CLASS and

CLASS - ATTRIBUTE - VALUE

OODBs have a schema (class) level and an instance (object) level. Thus, we need to consider two levels of generic patterns:

CLASS - RELATIONSHIP - CLASS and

INSTANCE - RELATIONSHIP - INSTANCE

**3.4.3.1 Examples of Patterns** In the following examples, every term within <brackets> marks a search term. Every term with no brackets is to be replaced by a literal term from the vocabulary. In OOHVR, each class in the OODB is referred to as an *Area* and each instance referred to as a *Term*.

1. <Area> : Lists all the areas in the InterMED.
2. Term - <Rel> : Lists all the relationships that emanate from a particular term.
3. Area - Rel : Returns true if the given area has that particular relationship.
4. Area - <Attrib> : Lists all the attributes of the given area.
5. Term1 - <Rel> - <Term2> : List the relationships that emanate from a particular term and also the terms to which those relationships point.
6. <Area1> - <Rel> - <Area2> : List all the area level triples.
7. Term - <Attrib> - <Value> : List all the attributes and their values for a particular term.

# CHAPTER 4

# API DESIGN

## 4.1 API Software Architecture

By definition, an API has to interact with a particular resource on one side and with various programs on the other side.

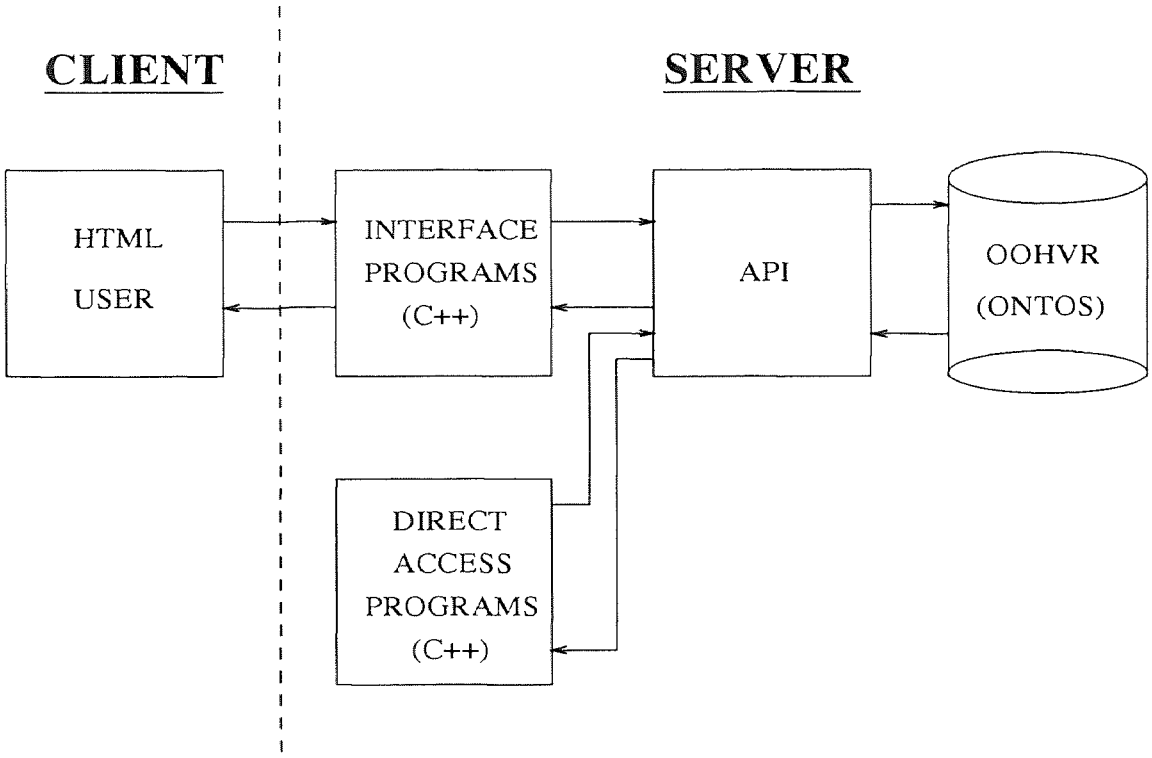


**Figure 4.1** API Software Architecture

As shown in Figure 4.1, for OOHVR the resource is a ONTOS OODB and the programs are the ones that provide user interfaces. These programs could be of two types [29]. Programs of the first type provide the form based and graphical user interfaces and can be referred to as the *"Interface Programs"*. Programs of the second type provide the programmatic interface and can be referred to as the *"Direct*

*Access Programs*". It should be noted that the interface programs need to execute on the server component of the software system since they share free store with the API functions.

## 4.2 API Specification

As discussed earlier, an analysis of the vocabulary structure has provided us with an exhaustive list of query patterns. The API specification aims at transforming these query patterns into function specifications; with a view to implementing these functions in the C++ programming language.

### 4.2.1 Additional Functions

While determining the set of access functions to be included in the API, a need for additional functions not directly identified by the query patterns was felt. These functions are required to take care of the following cases:

1. Distinction between local and inherited properties.
2. Basic composite operations – siblings, children etc.
3. Determination of the correlation between specific areas and terms. For example, which area does a particular term belong to?

### 4.2.2 Area and Term Level Functions

The vocabulary, implemented as an OODB, has schema level information as well as instance level information [10]. The schema information presents the area level hierarchy and the introduction of properties in different areas. This information is useful in comprehension of the vocabulary and understanding its modeling. The instance level presents the numerous terms in the vocabulary. Specific information about term level hierarchies, term properties and correlation between terms and areas can be obtained at this level.

Recognizing the significance of separate queries at these two levels of the database, each query pattern has been mapped to two separate (area and term level) functions providing similar functionality at the two levels.

### 4.2.3 Naming the Functions

The process of converting the query patterns into access functions begins by assigning names for each function. The name is based on the functionality of the query pattern corresponding to that particular function. The function names are long enough to clearly describe the function's purpose. This also provides unique names for the functions.

Further, each API function's name and each identifier in the API has the prefix *"MV"* (Medical Vocabulary). This ensures that the name of any function does not coincide with the name of another function in the relevant software environment.

### 4.2.4 Arguments to the Functions

The API functions accept zero to three strings as input, depending on the number of literals (constants) in the corresponding query pattern. The variables in the query patterns correspond to the output arguments for the function. In most cases, these output arguments are strings or pairs of strings. Also, since the number of output strings is a variable, a count for each set of strings is required. In some cases the output is more complex. Lists of strings may be nested inside another list of strings. For example, consider the function that accepts a term's name as input and returns a list of all relationships that emanate from it, along with the names of all the terms that those relationships point to. The output of this function will consist of a list of the relationships and a list of destination terms for each relationship (since a relationship can be multi-valued).

Special classes have been defined to encapsulate the input and output arguments' data for the API functions [3]. Each argument class typically contains

the input or output data item(s) and some member functions to access and modify them. The argument(s) to the functions will be objects of these argument classes. The creation of these special classes has the following advantages:

- The number of arguments to each function is reduced since we don't require separate arguments for a count of the strings in an output set.
- It is easier to handle the input and output data. The API programmer as well as the user interface programmer use the argument object's data access functions, without worrying about the actual data structure implementation.
- The actual internal implementation of the set(s) of strings and other data in an argument object may be altered without modifying the API functions or the user interface programs.
- It is easy to perform any processing on the data encapsulated in the argument objects. For example, it is possible to add a member function that provides the output strings in a HyperText Markup Language (HTML) compatible format.

### 4.2.5 Error Handling

Each API function returns an error, if any, by embedding an error code in an error object. The error class corresponding to this object facilitates error handling in the API by encapsulating the error code. Further, default error handling corresponding to each error code is provided as a member function of this error class. Overloading this function or bypassing it altogether enables an interface programmer to handle errors as per his or her requirements.

## 4.3 Example of API Specification

MV_API_Errors* **MV_List_All_Relationship_TargetTerm_Pairs_of_SourceTerm**(
MV_TermName &Source_Term,
MV_RelTermPairs &aRelTermList)

**Input:**

- *Source_Term*: A reference to a user defined object of the class MV_TermName which includes the following data member:
  - * TermName: The term from which the relationship(s) emanate.

**Output:**

- *aRelTermList*: A reference to a user defined object of the class MV_RelTermPairs which includes the following data members:
  - * Pairs: An array of pairs of relationships and destination terms. Pairs is of RelTerm_type, defined as:

```
typedef struct RelTerm
  {
   char *Relationship;
   char *Term;
  } RelTerm_type;
```

  - * No_of_Pairs: The number of relationship-term pairs detected.

**Description:**

This function returns a list of relationship(s) emanating from a particular term, the term(s) that the relationship(s) point to and a count of these relationship-term pairs.

## 4.4 Sharing the API Functions

There are many different user interface programmers accessing the API. Each of them has his or her own view and utilization of the vocabulary database. Yet, a uniform access to the API functions needs to be provided. Two factors need to be considered while designing this common access to the API functions. First, simultaneous access to the various API functions could be required. Second, the overhead associated with invoking an API function to access the vocabulary should be minimized.

The optimal solution for this problem is to create a shareable library that contains the object code for all the API functions. The UNIX operating system will handle and schedule all simultaneous accesses to this library [1, 2]. The API users will link to this shareable library in order to access one or more API functions.

# CHAPTER 5

# API IMPLEMENTATION

Based on the analysis of the OOHVR and the API design, a majority of retrieval functions have been implemented. The various implementation principles followed and issues involved are discussed in this chapter.

## 5.1 Vocabulary Access

As mentioned earlier, the InterMED medical vocabulary has been implemented as an Object-Oriented Database using the commercially available ONTOS OODB. To access the knowledge stored in this vocabulary, the API functions use object database constructs provided by ONTOS. For example, if an API function needs to locate a particular term in the database, it uses the index search iterator provided by ONTOS to search the index of all term names.

To use these ONTOS constructs, each API function's source code will have to include the appropriate header files. The ONTOS constructs' header files are stored in a fixed directory on the system. Hence, each API function will simply include the relevant header files from this directory.

Most API functions require details about the classes defined in the object database. For our vocabulary, these class declarations and definitions are obtained from the *Terms.h* and *Terms.C* files, respectively.

### 5.1.1 Retrieving Area Level Information

All the area level API functions need details regarding the schema of the object database. ONTOS provides a facility to retrieve schema level information from its database by providing descriptions of all the persistent classes in the database. All the class declarations are provided as objects in the database [24].

The name of such a persistent class description object is the same as the name of the corresponding class. Thus, when an API function requires information regarding a particular area in the vocabulary, it simply retrieves the ONTOS object with that area's name. The properties of that area are available in this class description.

ONTOS also provides facilities to retrieve hierarchical information for these schema level classes. Thus, given a particular area's persistent class description object, it is possible to determine the parent and child areas for that particular area.

### 5.1.2 Retrieving Term Level Information

Each term in the vocabulary has been stored as an object in the database. In the current implementation of the ONTOS database, these objects are not stored with a unique name. That is, none of these term objects can be identified or retrieved by just knowing its name. Instead, an index has been created that stores the name of a term and the corresponding object identifier. When an API function needs to retrieve a particular term's information, it uses this index to get the corresponding object identifier [24].

Some API functions require names of the properties (attributes and relationships) for a particular term. This translates into a query to the schema level of the database, although it is initiated through some term. Thus, the area corresponding to the given term is determined and the property names are obtained from that area's class description object.

The values for the properties of the term can be retrieved by using the object identifier. Attribute values can be obtained by using specific methods defined for that term. Relationship values, which represent the links to other terms in the vocabulary, typically provide names of the terms that the relationships point to. To obtain additional information regarding the destination terms, the relevant objects have to be retrieved separately.

It should be noted that within an area, the various terms are also related hierarchically. This follows from the semantic network, the predecessor of our object-oriented vocabulary. Since the concepts in the semantic network are not grouped in any manner, each of them has its own position in the network. An equivalent hierarchical position has been assigned to each term in the object-oriented vocabulary. To retrieve this term level hierarchy knowledge, relationships like SUBCLASS_OF and SUPERCLASS_OF are utilized.

## 5.2 Auxiliary Information about the Vocabulary

Many API functions require some auxiliary information about the vocabulary, primarily to simplify their algorithms and improve overall efficiency. Such information has been added to the database and is accessible to all API functions.

### 5.2.1 Relationship and Attribute Sets

When the properties for a particular area or term are retrieved from the schema of the database, it is not directly possible to classify them as attributes and relationships. To perform this classification, separate lists of names of all possible attributes and relationships in the vocabulary are needed. Then, for the area or term of interest, each property's name can be compared with the entries in these lists to determine whether that property is an attribute or a relationship.

Since this classification of properties into attributes and relationships is required quite frequently, two auxiliary sets have been added to the database:

1. The *RelationshipSet* contains the names of all the relationships in the vocabulary

2. The *AttributeSet* contains the names of all the attributes in the vocabulary

**5.2.1.1 Obtaining Lists of Attributes and Relationships** The task of determining all the attributes and relationships in a vocabulary is difficult and error prone if one tries to do it manually. For the InterMED vocabulary, this data was available in the form of flat files. These flat files represent the information stored in the semantic network version of the vocabulary. It should be noted that these same flat files were the starting point for the generation of the object database version of the vocabulary.

A *Gnu Awk* program is used to extract the names of all the properties from these flat files (Use the *gawk* command to execute this program [1].) A special identifier is also extracted. It enables the differentiation between an attribute and a relationship. This extracted information is stored in a file named *Properties.db*.

**5.2.1.2 Creating and Storing the Sets** A C++ program has been written to create and store the attribute and relationship sets in the database. This program looks at each property in the *Properties.db* file and uses the special identifier of that property to decide whether it is an attribute or a relationship. Accordingly, that property is stored in the attributes set or the relationships set. To simplify access to these sets (for all API functions), they have been stored in the database with specific names - *AttributeSet* and *RelationshipSet*. Thus, any function that uses these sets simply retrieves them by using the appropriate names [24].

## 5.3 Argument Classes

As explained in the previous chapter, the arguments to the API functions are encapsulated in special classes defined for this purpose. These are the argument classes. Each argument class typically contains the data item(s) needed by the API function and some access functions to create and modify these data item(s). These classes

have to ensure that the user interface programs as well the API functions can access the data item(s) without knowing the internal implementation.

The declarations of these argument classes are present in the *MV_API_IO.h* file and the definitions of various member functions are present in the *MV_API_IO.C* file. The member functions are typically concerned with maintaining the data structure(s) for the various data item(s) in that class.

Each API function includes the *MV_API_IO.h* header file in order to access the objects of the argument classes that are passed as arguments to the function. To utilize the member functions of the argument objects, each API function links to the object code for the argument classes - *MV_API_IO.o*.

## 5.4 Implementation of Error Handling

As discussed earlier, each API function returns a pointer to an object of an error class. This error class contains an error code and different error codes refer to the different errors that can occur when an API function is invoked. A list of the error codes and their meanings is given below.

1. NO_ERROR : The API function has been executed successfully.
2. ERR_INPUT : The input(s) to the API function is (are) invalid.
3. ERR_DATA_ABSENT : The requested data is not available in the vocabulary.
4. ERR_OPEN_FAIL : An attempt to open the database has failed.
5. ERR_DBNAME_ABSENT : The DBNAME environment variable has not been set.

Since the API functions return a pointer, a NULL return value can be used to represent successful completion of the function. This also facilitates the invocation of the API function within an "If" statement and immediate error handling, if required.

The default error handling for the various error codes has been provided in the form of a member function for the error class. This default error handling function displays the appropriate error message and terminates execution of the API function. If any specialized error handling is required, this member function can be overloaded or a separate function can be created to handle the error.

## 5.5 API Library Implementation

The API library primarily contains the object code for all the API functions. It also contains other object code (such as MV_API_IO.o - the argument classes implementation code) to be used by more than one API function. Since the entire API is to be exported using this library, it is necessary to ensure that the library can be moved around (either locally on the same system or even to an entirely new system).

The UNIX based *"archive"* facility satisfies all our requirements [2]. This facility permits the creation and maintenance of a library of object codes. The UNIX archive command *"ar"* has various command line switches that provide the required functionality. Examples of using this command:

- ar -q libMVAPI.a api_func1.o ==> Quick append the object code api_func1.o to the archived library MVAPI
- ar -r libMVAPI.a api_func2.o ==> Replace the library's existing object code for api_func3 with the new object code specified
- ar -d libMVAPI.a api_func3.o ==> Delete the object code api_func3.o from the library MVAPI
- ar -vt libMVAPI.a ==> List the contents of the library MVAPI

## 5.6 API Implementation Architecture

Each API function uses ONTOS constructs, vocabulary implementation details and argument classes to produce the API library. This API library is accessible to the different user interface programmers. This interaction between various components of the system is depicted in Figure 5.1.

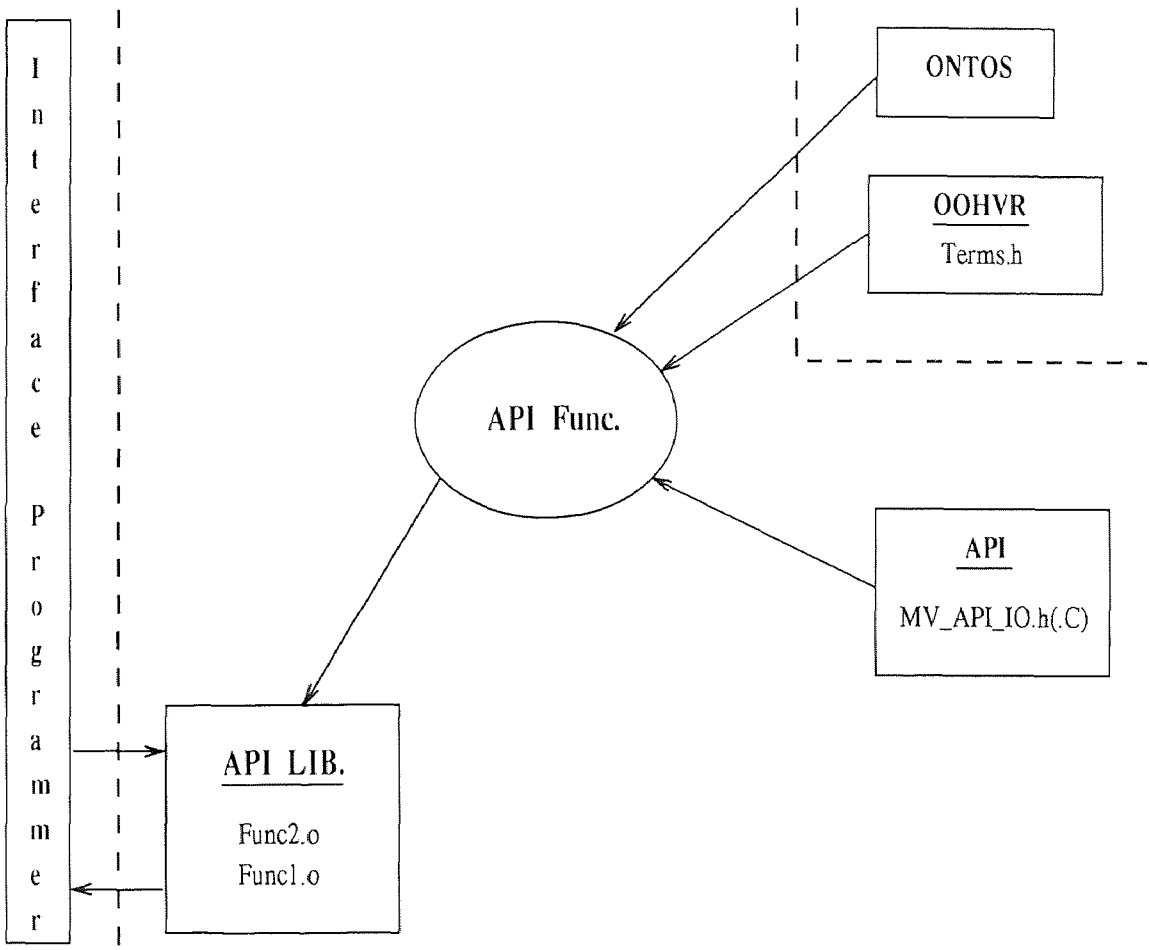


**Figure 5.1** API Implementation Architecture

## 5.7 Miscellaneous Implementation Issues

### 5.7.1 Accessing Different Databases

The API functions can be used for object database implementations of different medical vocabularies. Hence, they must possess the flexibility to access different databases without requiring any changes in their own implementation. We have incorporated this flexibility by using a special environment variable, DBNAME, to indicate the name of the database to be accessed by the API functions. Each API function uses the *getenv()* system call [2] to obtain the value of this environment variable. This value is assumed to be the name of the database that the API function should access.

### 5.7.2 Making each Transaction more Efficient

Each API function has at least one database transaction. The data retrieval functions do not make any changes to the database, hence it is not necessary to commit their transactions to the database. By aborting transactions after the required data has been retrieved, the much more expensive database commit operation is skipped. This improves the overall efficiency, per transaction, of the data retrieval API functions.

# CHAPTER 6

# CONCLUSIONS AND FUTURE WORK

Controlled vocabularies serve as excellent tools for the management of diverse terminologies within an application field. In this thesis, we have described a standardized interface (the API) for users of a vocabulary implemented as an OODB system. This interface provides efficient retrieval of information available in the vocabulary. Since the outputs of this interface are basically strings and numbers, they can be used by various user interface programs. While our discussions were centered around a medical vocabulary that we have implemented, the techniques described are readily applicable to any vocabulary modeled as an OODB system. We also presented the architecture and implementation of a software system that provides the standardized interface. Presently this interface is up and running for the InterMED vocabulary within the context of the ONTOS OODB system. A web-based user interface that utilizes the API has also been implemented by a peer group.

Further development steps for the API have been identified. These steps primarily deal with new functions for additional vocabulary access.

1. **Updating the OOHVR:** Since the vocabulary is not static, functions to modify, add and delete information will be required. These updates could apply to the properties of existing terms and areas or result in the addition of new areas and terms. Thus there are two kinds of updates: updates of the data and updates of the schema. Due to the mechanisms that are used to generate the schema, there are situations when even an update of the data results in the necessity of a schema change.

   The issues involved in updating the vocabulary have been identified and documented. The specification and implementation of the API functions

that perform these updates will be the next step in the development of the standardized interface.

2. **Versions of the Vocabulary Terms:** The terminology in the medical field changes from time to time. For example, the same drug could have different names during different time spans. Yet, a query to the vocabulary may refer to an older name or may require results corresponding to terminology used in a specific time span. This implies that different versions of the terms in the vocabulary will have to coexist. Also, we will need information that indicates how different versions of a term are related to each other.

   API functions will be required to provide and maintain this versioning capability for the vocabulary.

3. **Pattern Matching:** Presently, precise string inputs need to be provided to the API functions. This is very cumbersome, especially in the medical field due to the long and complex terminology used. Pattern matching will attempt to allow users to provide only partial or abbreviated strings as inputs.

   API functions capable of mapping partial or abbreviated inputs to their appropriate full length versions need to be developed.

# APPENDIX A

# MISCELLANEOUS DETAILS

Certain aspects of the API implemenation and maintenance will be discussed here.

## A.1 Compile and Link Procedures

We use the ONTOS *cplus* compiler to compile and link each API function. This compiler pre-processes the ONTOS constructs in a source file and converts them to equivalent C++ code. This entire code is then passed on to the standard CC C++ compiler. This mechanism permits the use of the same command line switches for cplus as those available for CC.

### A.1.1 Compiling

The following alias is used to compile each API function (and the program invoking the API function(s)).

```
ac+ : cplus -c -g !* -I. -I/opt/ONTOS/include -I/home/geller/OOHVR \
      /limin/V31/v2 -I/home/geller/OOHVR/API/include -DSUNSPRO\_CC \
      -D\_NO\_LONGLONG
```

Each component of this compile command is explained below:

- *-c :* Standard CC compiler switch to indicate that only compilation of the specified source files is required.
- *-g :* Standard CC compiler switch to indicate that debugging information should be included in the object files created by the compilation.
- *!* :* Represents the names of all the source files specified.
- *-I. :* This switch asks the compiler to search the current directory for header files included in the source code.

- *-I/opt/ONTOS/include :* Asks compiler to search this directory for header files. We have the ONTOS constructs related header files in this directory.

- *-I/home/geller/OOHVR/limin/V31/v2 :* The OOHVR related Terms.h header file is in this directory, so we ask the compiler to look up this directory.

- *-I/home/geller/OOHVR/API/include :* The API argument class definition file (MV_API_IO.h) and other shareable files are in this directory.

  This compilation alias can be used as shown below:

```
ac+ MV\_List\_All\_Children\_of\_Term.C
```

  where,

  MV_List_All_Children_of_Term.C is the name of the source file to be compiled (It contains the source code for the MV_List_All_Children_of_Term API function).

### A.1.2 Linking

The following alias is used to link each API function (and the program invoking the API function(s)).

```
ac+2 : cplus -g !* -L/opt/ONTOS/lib -lONTOS -L/home/geller/OOHVR \
       /API/lib -lMVAPI -lCNet -lsocket -lnsl -lelf
```

Each component of this compile command is explained below:

- *-g :* Standard CC compiler switch to indicate that debugging information should be included in the object files created by the compilation.

- *!* :* Represents the names of all the object files specified.

- *-L/opt/ONTOS/lib :* The -L switch defines a directory to search for libraries required during the link and load procedure. Here, a path is set for the ONTOS library.

- *-lONTOS :* Specifies linkage to the ONTOS library.
- *-L/home/geller/OOHVR/API/lib :* Sets the directory path for the API library.
- *-lAPI :* Specifies linkage to the API library. This enables an API function to invoke other API functions already present in the library.

  This linkage alias can be used as shown below:

```
ac+2 MV\_List\_All\_Children\_of\_Term.o \
     Prog\_MV\_List\_All\_Children\_of\_Term.o Terms.o
```

  where,

  MV_List_All_Children_of_Term.o is the name of the API function's object file.

  Prog_MV_List_All_Children_of_Term.o is the name of the test program for the API function.

  Terms.o is the name of the object file for the class definitions of the object database for our vocabulary.

The *Terms.o* object code has to be included here because ONTOS requires explicit linkage to the schema definition code. Hopefully, ONTOS will be able to overcome this technical hurdle.

## A.2 Directory Structure

To organize all data, files etc. a specific directory structure has been created.

We start with the directory */home/geller/OOHVR/API* and treat this as the home directory for the API.

The following directories have been created under the API directory:

- *include :* Contains all the header files to be included by different API functions. (e.g. MV_API_IO.h)

- *lib :* Contains the MVAPI library.
- *Funcs :* The source code for the functions is stored in this directory.
- *Progs :* The source code of test programs for the API functions are stored in this directory.
- *bin :* The object code for the functions is stored in this directory.

Each directory contains a README file explaining the purpose of that particular directory.

### A.3 Important Files

Some files have been created and maintained with a view to better communication amongst team members and efficient project management.

- */home/geller/OOHVR/API/assigns.doc :* This file is used for task assignment to the various API group team members. It contains a list of all API functions' names and the name(s) of the individual(s) assigned to code each function.
- */home/geller/OOHVR/API/Storing.doc :* This file describes the steps involved in coding an API function and then storing it and the relevant test program in the appropriate directories.
- */home/geller/OOHVR/API/API_List.doc :* This file contains entries for all the completed API functions. It represents the current status of the API group. A new entry in this file also indicates a need to add that API function's object code to the MVAPI library.
- */home/geller/OOHVR/API/specific_V2_msword.uu :* This uuencoded file contains the specification for API functions. The contents are in the MS-Word document format.

# APPENDIX B

# CODE FOR THE API

The code for the following components of the API is presented here:

- Argument Classes
- Function Implementation

## B.1 Argument Classes

### B.1.1 The Class Declarations

```
/*********************************************************************
 * MV_API_IO.h : This header file contains all the class declarations
 * required for the parameter objects of the API. It should be
 * included by all the programs that access the API.
 *
 * Created by : Hemant Kothavade          Creation Date : 6/3/96
 * Last Updated : 9/20/96
 *********************************************************************/

/*********************************************************************
 * Design Notes :
 *
 * 1.  We have designed a separate class for each type of parameter
 *     needed instead of creating a hierarchy. This is to enable
 *     each parameter class's members to have unique names related
 *     to their functionality.
 *
 * 2.  The 'MV' prefix to each class name denotes 'Medical
 *      Vocabulary'.
 *********************************************************************/

/*********************************************************************
 * Implementation Notes :
 *
 * 1.  The #defines for ROOT_AREA etc. are specific to a database.
 *     These should be changed when the API is to be used for a
 *     different database.
```

```
 *
 ************************************************************************/

#include <string.h>

// Defines to be used in the API functions.

// This corresponds to the name of the root area in the OOHVR for
// which the API is being used
#define ROOT_AREA ENTITY_AREA

// This is the name of the property that provides the name of each
// area and term stored in the OOHVR
#define ID_PROPERTY NAME

// Required during a OC_lookup etc.
#define STR_ROOT_AREA "ENTITY_AREA"

// Required during a OC_lookup etc.
#define STR_ID_SEARCH "ENTITY_AREA::NAME"

// The set with names of all relationships.
#define STR_RELATIONSHIP_SET "RelationshipSet"

// The set with names of all attributes.
#define STR_ATTRIBUTE_SET "AttributeSet"

#define SHADOW_CONCAT "_P"

#define AREA_SUFFIX_CONCAT "_AREA"



// Global declarations to be used by the API functions.
enum bool_type {FALSE, TRUE};

/*  The classes for the input parameters */

class MV_TermName
{
  private :
    char *TermName;

  public :
```

```
    MV_TermName()
    {
      TermName = NULL;
    }
    MV_TermName(char *init_name)
      {
TermName = new char[strlen(init_name) + 1];
strcpy(TermName, init_name);
      }
    void setTermName(char *iname)
      {
delete TermName;
TermName = new char[strlen(iname) + 1];
strcpy(TermName, iname);
      }
    char *getTermName()
      {
return (TermName);
      }
    ~MV_TermName()
      {
delete TermName;
      }
};


class MV_AreaName
{
  private :
    char *AreaName;

  public :
    MV_AreaName()
    {
      AreaName = NULL;
    }
    MV_AreaName(char *init_name)
      {
AreaName = new char[strlen(init_name) + 1];
strcpy(AreaName, init_name);
      }
    void setAreaName(char *iname)
      {
delete AreaName;
```

```
AreaName = new char[strlen(iname) + 1];
strcpy(AreaName, iname);
      }
    char *getAreaName()
      {
return (AreaName);
      }
    ~MV_AreaName()
      {
delete AreaName;
      }
};


class MV_RelationshipName
{
  private :
    char *RelName;

  public :
    MV_RelationshipName()
    {
      RelName = NULL;
    }
    MV_RelationshipName(char *init_name)
      {
RelName = new char[strlen(init_name) + 1];
strcpy(RelName, init_name);
      }
    void setRelName(char *iname)
      {
delete RelName;
RelName = new char[strlen(iname) + 1];
strcpy(RelName, iname);
      }
    char *getRelName()
      {
return (RelName);
      }
    ~MV_RelationshipName()
      {
delete RelName;
      }
};
```

```
class MV_AttributeName
{
  private :
    char *AttrName;

  public :
    MV_AttributeName()
    {
      AttrName = NULL;
    }
    MV_AttributeName(char *init_name)
      {
AttrName = new char[strlen(init_name) + 1];
strcpy(AttrName, init_name);
      }
    void setAttrName(char *iname)
      {
delete AttrName;
AttrName = new char[strlen(iname) + 1];
strcpy(AttrName, iname);
      }
    char *getAttrName()
      {
return (AttrName);
      }
    ~MV_AttributeName()
      {
delete AttrName;
      }
};


class MV_Value
{
  private :
    char *Value;

  public :
    MV_Value()
    {
      Value = NULL;
    }
```

```
    MV_Value(char *init_value)
      {
Value = new char[strlen(init_value) + 1];
strcpy(Value, init_value);
      }
    void setValue(char *ival)
      {
delete Value;
Value = new char[strlen(ival) + 1];
strcpy(Value, ival);
      }
    char *getValue()
      {
return (Value);
      }
    ~MV_Value()
      {
delete Value;
      }
};

/* ------------------------------------------------------------- */

/* The classes for the output parameters */


class MV_TermList
{
  private :
    char **TermList;
    int No_of_Terms;

    int term_count;              // No. of terms added so far

  public :
    MV_TermList()
    {
      No_of_Terms = -1;
      term_count = -1;
    }
    void createList(int nterms)    // nterms in the list
      {
 No_of_Terms = nterms;
 TermList = (char **) new char *[No_of_Terms];
```

```
      }
    void addTerm(char *aTerm);    /* In MV_API_IO.C  */

    /* Use only to reduce the term count, if needed */
    void  setTermCount(int new_count)
      {
No_of_Terms = new_count;
      }
    char **getTermList()
      {
return TermList;
      }
    int  getTermCount()
      {
return No_of_Terms;
      }
    ~MV_TermList()
      {
for(int i = 0; i < No_of_Terms; i++)
  delete TermList[i];
delete [] TermList;
      }
};


class MV_AreaList
{
  private :
    char **AreaList;
    int No_of_Areas;

    int area_count;              // No. of areas added so far

  public :
    MV_AreaList()
    {
      No_of_Areas = -1;
      area_count = -1;
    }
    void createList(int nareas)    // nareas in the list
      {
No_of_Areas = nareas;
AreaList = (char **) new char *[No_of_Areas];
      }
```

```
    void addArea(char *anArea);

    /* Use only to reduce the area count, if needed */
    void  setAreaCount(int new_count)
      {
No_of_Areas = new_count;
      }
    char **getAreaList()
      {
return AreaList;
      }
    int  getAreaCount()
      {
return No_of_Areas;
      }
    ~MV_AreaList()
      {
for(int i = 0; i < No_of_Areas; i++)
  delete AreaList[i];
delete [] AreaList;
      }
};


class MV_RelationshipList
{
  private :
    char **RelList;
    int No_of_Rels;

    int rel_count;              // No. of rels added so far

  public :
    MV_RelationshipList()
    {
      No_of_Rels = -1;
      rel_count = -1;
    }
    void createList(int nrels)       // nrels in the list
      {
No_of_Rels = nrels;
RelList = (char **) new char *[No_of_Rels];
      }
    void addRelationship(char *aRel);
```

```
      /* Use only to reduce the rel count, if needed */
      void  setRelationshipCount(int new_count)
        {
 No_of_Rels = new_count;
        }
      char **getRelationshipList()
        {
 return RelList;
        }
      int  getRelationshipCount()
        {
 return No_of_Rels;
        }
      ~MV_RelationshipList()
        {
 for(int i = 0; i < No_of_Rels; i++)
   delete RelList[i];
 delete [] RelList;
        }
};


class MV_AttributeList
{
  private :
    char **AttrList;
    int No_of_Attrs;

    int attr_count;              // No. of attrs added so far

  public :
    MV_AttributeList()
    {
      No_of_Attrs = -1;
      attr_count = -1;
    }
    void createList(int nattrs)      // nattrs in the list
      {
No_of_Attrs = nattrs;
AttrList = (char **) new char *[No_of_Attrs];
      }
    void addAttribute(char *anAttr);
```

```
    /* Use only to reduce the attr count, if needed */
    void  setAttributeCount(int new_count)
      {
No_of_Attrs = new_count;
      }
    char **getAttributeList()
      {
return AttrList;
      }
    int  getAttributeCount()
      {
return No_of_Attrs;
      }
    ~MV_AttributeList()
      {
for(int i = 0; i < No_of_Attrs; i++)
  delete AttrList[i];
delete [] AttrList;
      }
};


class MV_ValueList
{
  private :
    char **ValList;
    int No_of_Vals;

    int val_count;              // No. of vals added so far

  public :
    MV_ValueList()
    {
      No_of_Vals = -1;
      val_count = -1;
    }
    void createList(int nvals)     // nvals in the list
      {
No_of_Vals = nvals;
ValList = (char **) new char *[No_of_Vals];
      }
    void addValue(char *aVal);

    /* Use only to reduce the value count, if needed */
```

```
    void  setValueCount(int new_count)
      {
No_of_Vals = new_count;
      }
    char **getValueList()
      {
return ValList;
      }
    int  getValueCount()
      {
return No_of_Vals;
      }
    ~MV_ValueList()
      {
for(int i = 0; i < No_of_Vals; i++)
  delete ValList[i];
delete [] ValList;
      }
};


class MV_TermRelPairs
{
  private :
    typedef struct TermRel    // Each Term will have one or
    {                         // more relationships.
      char *Term;
      char **Relationships;
      int No_of_Rels;         // Total rels for this Term
    }TermRel_type;

    TermRel_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of Terms retrieved

    int pair_count;           // No. of pairs entered so far
    int rel_count;     // No. of rels so far for particular Term

  public :
    MV_TermRelPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      rel_count = -1;
    }
```

```
    void createList(int npairs)  // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (TermRel_type *) new TermRel_type[No_of_Pairs];
      }
    void addTerm(char *aTerm, int nrels);
    void addRelationship(char *aTerm, char *aRel);
    int  getTermRelPairCount()
      {
return No_of_Pairs;
      }

    void getTermList(char **&aTermList, int &no_of_terms);
// Use the following function in a loop to get each term and its
// relationships OR
// Use the position in the term list to get relationships of a
// particular term
    int getTermRelationships(int pos, char *&aTerm, char **&aRelList,
                             int &no_of_rels);
    ~MV_TermRelPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete (Pairs[i].Term);
    for (int j = 0; j < Pairs[i].No_of_Rels; j++)
      delete Pairs[i].Relationships[j];
    delete [] Pairs[i].Relationships;
  };
delete [] Pairs;
      }
};


class MV_AreaRelPairs
{
  private :
    typedef struct AreaRel    // Each Area will have one or
    {                          // more relationships.
      char *Area;
      char **Relationships;
      int No_of_Rels;         // Total rels for this Area
    }AreaRel_type;

    AreaRel_type *Pairs;   // List of pairs
```

```
    int No_of_Pairs;          // Equals no. of Areas retrieved

    int pair_count;           // No. of pairs entered so far
    int rel_count;     // No. of rels so far for particular Area

  public :
    MV_AreaRelPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      rel_count = -1;
    }
    void createList(int npairs)    // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (AreaRel_type *) new AreaRel_type[No_of_Pairs];
      }
    void addArea(char *anArea, int nrels);
    void addRelationship(char *anArea, char *aRel);
    int  getAreaRelPairCount()
      {
return No_of_Pairs;
      }

    void getAreaList(char **&anAreaList, int &no_of_areas);
// Use the following function in a loop to get each area and its
// relationships OR
// Use the position in the area list to get relationships of a
// particular area
    int getAreaRelationships(int pos, char *&anArea,
                             char **&aRelList, int &no_of_rels);
    ~MV_AreaRelPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Area;
    for (int j = 0; j < Pairs[i].No_of_Rels; j++)
      delete Pairs[i].Relationships[j];
    delete [] Pairs[i].Relationships;
  };
delete [] Pairs;
      }
};
```

```
class MV_TermAttrPairs
{
  private :
    typedef struct TermAttr    // Each Term will have one or
    {                          // more attributes.
      char *Term;
      char **Attributes;
      int No_of_Attrs;         // Total attrs for this Term
    }TermAttr_type;

    TermAttr_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of Terms retrieved

    int pair_count;           // No. of pairs entered so far
    int attr_count;     // No. of attrs so far for particular Term

  public :
    MV_TermAttrPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      attr_count = -1;
    }
    void createList(int npairs)     // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (TermAttr_type *) new TermAttr_type[npairs];
      }
    void addTerm(char *aTerm, int nattrs);
    void addAttribute(char *aTerm, char *anAttr);
    int  getTermAttrPairCount()
      {
return No_of_Pairs;
      }

    void getTermList(char **&aTermList, int &no_of_terms);
// Use the following function in a loop to get each term and its
// attributes OR
// Use the position in the term list to get attributes of a
// particular term
    int getTermAttributes(int pos, char *&aTerm, char **&anAttrList,
                          int &no_of_attrs);
    ~MV_TermAttrPairs()
```

```
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Term;
    for (int j = 0; j < Pairs[i].No_of_Attrs; j++)
      delete Pairs[i].Attributes[j];
    delete [] Pairs[i].Attributes;
  };
delete [] Pairs;
      }
};


class MV_AreaAttrPairs
{
  private :
    typedef struct AreaAttr    // Each Area will have one or
    {                          // more attributes.
      char *Area;
      char **Attributes;
      int No_of_Attrs;         // Total attrs for this Area
    }AreaAttr_type;

    AreaAttr_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of Areas retrieved

    int pair_count;           // No. of pairs entered so far
    int attr_count;     // No. of attrs so far for particular Area

  public :
    MV_AreaAttrPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      attr_count = -1;
    }
    void createList(int npairs)   // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (AreaAttr_type *) new AreaAttr_type[No_of_Pairs];
      }
    void addArea(char *anArea, int nattrs);
    void addAttribute(char *anArea, char *anAttr);
    int  getAreaAttrPairCount()
```

```
      {
return No_of_Pairs;
      }

    void getAreaList(char **&anAreaList, int &no_of_areas);
// Use the following function in a loop to get each area and its
// attributes OR
// Use the position in the area list to get attributes of a
// particular area
    int getAreaAttributes(int pos, char *&anArea,
                          char **&anAttrList, int &no_of_attrs);
    ~MV_AreaAttrPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Area;
    for (int j = 0; j < Pairs[i].No_of_Attrs; j++)
      delete Pairs[i].Attributes[j];
    delete [] Pairs[i].Attributes;
  };
delete [] Pairs;
      }
};


class MV_RelTermPairs
{
  private :
    typedef struct RelTerm    // Each relationship of a Term will
    {                         // point to a particular Term
      char *Relationship;
      char *Term;
    }RelTerm_type;

    RelTerm_type *Pairs;   // List of pairs
    int No_of_Pairs;      // Equals no. of relationships for the Term

    int pair_count;           // No. of pairs entered so far

  public :
    MV_RelTermPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
```

```
    }
    void createList(int npairs)    // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (RelTerm_type *) new RelTerm_type[No_of_Pairs];
      }
    void addRelTermPair(char *aRel, char *aTerm);
    int  getRelTermPairCount()
      {
return No_of_Pairs;
      }

// Use the following function in a loop to get each relationship and
// the term it points to
    int  getRelationshipTerm(int pos, char *&aRel, char *&aTerm);
    ~MV_RelTermPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Relationship;
    delete Pairs[i].Term;
  };
delete [] Pairs;
      }
};


class MV_RelAreaPairs
{
  private :
    typedef struct RelArea    // Each relationship of a Area will
    {                         // point to a particular Area
      char *Relationship;
      char *Area;
    }RelArea_type;

    RelArea_type *Pairs;     // List of pairs
    int No_of_Pairs;      // Equals no. of relationships for the Area

    int pair_count;           // No. of pairs entered so far

  public :
    MV_RelAreaPairs()
    {
```

```
      No_of_Pairs = -1;
      pair_count = -1;
    }
    void createList(int npairs)   // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (RelArea_type *) new RelArea_type[No_of_Pairs];
      }
    void addRelAreaPair(char *aRel, char *anArea);
    int  getRelAreaPairCount()
      {
return No_of_Pairs;
      }

// Use the following function in a loop to get each relationship and
// the area it points to
    int  getRelationshipArea(int pos, char *&aRel, char *&anArea);
    ~MV_RelAreaPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Relationship;
    delete Pairs[i].Area;
  };
delete [] Pairs;
      }
};


class MV_TermPairs
{
  private :
    typedef struct Terms
    {
      char *Source_Term;
      char *Dest_Term;
    }Term_type;

    Term_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of Term pairs detected

    int pair_count;           // No. of pairs entered so far

  public :
```

```
    MV_TermPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
    }
    void createList(int npairs)  // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (Term_type *) new Term_type[No_of_Pairs];
      }
    void addTermPair(char *aSourceTerm, char *aDestTerm);
    int  getTermPairCount()
      {
return No_of_Pairs;
      }
    int  getTerms(int pos, char *&aSourceTerm, char *&aDestTerm);
    ~MV_TermPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Source_Term;
    delete Pairs[i].Dest_Term;
  };
delete [] Pairs;
      }
};


class MV_AreaPairs
{
  private :
    typedef struct Areas
    {
      char *Source_Area;
      char *Dest_Area;
    }Area_type;

    Area_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of Area pairs detected

    int pair_count;           // No. of pairs entered so far

  public :
    MV_AreaPairs()
```

```
    {
      No_of_Pairs = -1;
      pair_count = -1;
    }
    void createList(int npairs)   // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (Area_type *) new Area_type[No_of_Pairs];
      }
    void addAreaPair(char *aSourceArea, char *aDestArea);
    int  getAreaPairCount()
      {
return No_of_Pairs;
      }
    int  getAreas(int pos, char *&aSourceArea, char *&aDestArea);
    ~MV_AreaPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Source_Area;
    delete Pairs[i].Dest_Area;
  };
delete [] Pairs;
      }
};


class MV_AreaTermPairs
{
  private :
    typedef struct AreaTerm    // Each Area will have one or
    {                          // more insatnces.
      char *Area;
      char **Terms;
      int No_of_Terms;         // Total Terms for this Area
    }AreaTerm_type;

    AreaTerm_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of Areas retrieved

    int pair_count;           // No. of pairs entered so far
    int term_count;     // No. of Terms so far for particular Area

  public :
```

```
    MV_AreaTermPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      term_count = -1;
    }
    void createList(int npairs)  // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (AreaTerm_type *) new AreaTerm_type[No_of_Pairs];
      }
    void addArea(char *anArea, int nterms);
    void addTerm(char *anArea, char *aTerm);
    int  getAreaTermPairCount()
      {
return No_of_Pairs;
      }

    void getAreaList(char **&anAreaList, int &no_of_areas);
// Use the following function in a loop to get each area and its
// terms OR
// Use the position in the area list to get terms of a particular
// area
    int getAreaTerms(int pos, char *&anArea, char **&aTermList,
                     int &no_of_terms);
    ~MV_AreaTermPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Area;
    for (int j = 0; j < Pairs[i].No_of_Terms; j++)
      delete Pairs[i].Terms[j];
    delete [] Pairs[i].Terms;
  };
delete [] Pairs;
      }
};


class MV_TermTriples
{
  private :
    typedef struct Terms
    {
```

```
      char *Source_Term;
      char *Relationship;
      char *Dest_Term;
    }Term_type;

    Term_type *Triples;   // List of triples
    int No_of_Triples;     // Equals no. of Term triples detected

    int triple_count;           // No. of triples entered so far

  public :
    MV_TermTriples()
    {
      No_of_Triples = -1;
      triple_count = -1;
    }
    void createList(int ntriples) // ntriples is total no. of triples
      {
No_of_Triples = ntriples;
Triples = (Term_type *) new Term_type[No_of_Triples];
      }
    void addTermTriple(char *aSourceTerm, char *aRelationship,
       char *aDestTerm);
    int  getTermTripleCount()
      {
return No_of_Triples;
      }
    int  getTermTriple(int pos, char *&aSourceTerm,
                       char *&aRelationship, char *&aDestTerm);
    ~MV_TermTriples()
      {
for (int i = 0; i < No_of_Triples; i++)
  {
    delete Triples[i].Source_Term;
    delete Triples[i].Relationship;
    delete Triples[i].Dest_Term;
  };
delete [] Triples;
      }
};


class MV_AreaTriples
{
```

```
  private :
    typedef struct Areas
    {
      char *Source_Area;
      char *Relationship;
      char *Dest_Area;
    }Area_type;

    Area_type *Triples;   // List of triples
    int No_of_Triples;       // Equals no. of Area triples detected

    int triple_count;           // No. of triples entered so far

  public :
    MV_AreaTriples()
    {
      No_of_Triples = -1;
      triple_count = -1;
    }
    void createList(int ntriples) // ntriples is total no. of triples
      {
No_of_Triples = ntriples;
Triples = (Area_type *) new Area_type[No_of_Triples];
      }
    void addAreaTriple(char *aSourceArea, char *aRelationship,
       char *aDestArea);
    int  getAreaTripleCount()
      {
return No_of_Triples;
      }
    int  getAreaTriple(int pos, char *&aSourceArea,
                       char *&aRelationship, char *&aDestArea);
    ~MV_AreaTriples()
      {
for (int i = 0; i < No_of_Triples; i++)
  {
    delete Triples[i].Source_Area;
    delete Triples[i].Relationship;
    delete Triples[i].Dest_Area;
  };
delete [] Triples;
      }
};
```

```
class MV_AttrValPairs
{
  private :
    typedef struct AttrVal    // Each attribute will have one or
    {                         // more values.
      char *Attribute;
      char **Values;
      int No_of_Vals;         // Total values for this attribute
    }AttrVal_type;

    AttrVal_type *Pairs;   // List of pairs
    int No_of_Pairs;          // Equals no. of attributes retrieved

    int pair_count;           // No. of pairs entered so far
    int val_count;    // No. of vals so far for particular Attribute

  public :
    MV_AttrValPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      val_count = -1;
    }
    void createList(int npairs)  // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (AttrVal_type *) new AttrVal_type[No_of_Pairs];
      }
    void addAttribute(char *anAttr, int nvals);
    void addValue(char *anAttr, char *aVal);
    int  getAttrValPairCount()
      {
return No_of_Pairs;
      }

    void getAttributeList(char **&anAttrList, int &no_of_attrs);
// Use the following function in a loop to get each attribute and
// its values OR
// Use the position in the attribute list to get values of a
// particular attribute
    int getAttributeValues(int pos, char *&anAttribute,
                           char **&aValList, int &no_of_vals);
    ~MV_AttrValPairs()
```

```
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Attribute;
    for (int j = 0; j < Pairs[i].No_of_Vals; j++)
      delete Pairs[i].Values[j];
    delete [] Pairs[i].Values;
  };
delete [] Pairs;
      }
};


class MV_TermValPairs
{
  private :
    typedef struct TermVal    // Each Term will have one or
    {                         // more values for the attribute.
      char *Term;
      char **Values;
      int No_of_Vals;       // Total values for this Terms attribute
    }TermVal_type;

    TermVal_type *Pairs;     // List of pairs
    int No_of_Pairs;          // Equals no. of Terms retrieved

    int pair_count;           // No. of pairs entered so far
    int val_count;     // No. of vals so far for particular Term

  public :
    MV_TermValPairs()
    {
      No_of_Pairs = -1;
      pair_count = -1;
      val_count = -1;
    }
    void createList(int npairs)  // npairs is total no. of pairs
      {
No_of_Pairs = npairs;
Pairs = (TermVal_type *) new TermVal_type[No_of_Pairs];
      }
    void addTerm(char *aTerm, int nvals);
    void addValue(char *aTerm, char *aVal);
    int  getTermValPairCount()
```

```
      {
return No_of_Pairs;
      }

    void getTermList(char **&aTermList, int &no_of_terms);
// Use the following function in a loop to get each term and
// the values OR
// Use the position in the term list to get values for a
//particular term
    int getTermValues(int pos, char *&aTerm, char **&aValList,
                      int &no_of_vals);
    ~MV_TermValPairs()
      {
for (int i = 0; i < No_of_Pairs; i++)
  {
    delete Pairs[i].Term;
    for (int j = 0; j < Pairs[i].No_of_Vals; j++)
      delete Pairs[i].Values[j];
    delete [] Pairs[i].Values;
  };
delete [] Pairs;
      }
};



class MV_TermAttrValTriples
{
  private :
    typedef struct TermAttrVal
    {
      char *Term;
      char *Attribute;
      char **Values;
      int No_of_Vals;
    }TermAttrVal_type;

    TermAttrVal_type *Triples;   // List of triples
    int No_of_Triples;         // Equals no. of Term triples detected

    int triple_count;           // No. of triples entered so far
    int val_count;

  public :
```

```
    MV_TermAttrValTriples()
    {
      No_of_Triples = -1;
      triple_count = -1;
      val_count = -1;
    }
    void createList(int ntriples) // ntriples is total no. of triples
      {
No_of_Triples = ntriples;
Triples = (TermAttrVal_type *) new TermAttrVal_type[No_of_Tri
ples];
      }
    void addTermAttribute(char *aTerm, char *anAttribute, int nvals);
    void addValue(char *aTerm, char *anAttribute, char *aValue);
    int  getTermTripleCount()
      {
return No_of_Triples;
      }
    int  getTermAttrValTriple(int pos, char *&aTerm,
                              char *&anAttribute,
                              char **&aValueList, int &no_of_vals);
    ~MV_TermAttrValTriples()
      {
for (int i = 0; i < No_of_Triples; i++)
  {
    delete Triples[i].Term;
    delete Triples[i].Attribute;
    for (int j = 0; j < No_of_Triples; j++)
      delete Triples[i].Values[j];
    delete [] Triples[i].Values;
  };
delete [] Triples;
      }
};


#define NO_ERROR 0
#define ERR_INPUT 1
#define ERR_DATA_ABSENT 2
#define ERR_OPEN_FAIL 3
#define ERR_DBNAME_ABSENT 4

class MV_API_Errors
{
```

```
  private :
    int Error_code;

  public :
    MV_API_Errors()
    {
      Error_code = 0;
    }
    MV_API_Errors(int ierr)
      {
Error_code = ierr;
      }
    void setErrorCode(int err)
      {
Error_code = err;
      }
    int  getErrorCode()
      {
return Error_code;
      }
    void HandleError();     // A case statement to handle all errors
};
```

### B.1.2 Implementation of Member Functions

```
/*********************************************************************
 * MV_API_IO.C : This file contains the implementation of the member
 * functions for the classes defined in MV_API_IO.h
 *
 * Created by : Hemant Kothavade Creation Date : 6/3/96
 * Last Updated : 6/13/96
 *********************************************************************/

/*********************************************************************
 * The 'MV' prefix denotes 'Medical Vocabulary'.
 *********************************************************************/

#include <iostream.h>
#include "MV_API_IO.h"


/*********************************************************************
 * Function Name : MV_TermList :: addTerm
 *
```

```
 * Inputs :-
 * 1. aTerm(Type char *) : The Term to be added in the list.
 *
 * Outputs :-
 * 1. An entry for the new Term name in the list TermList.
 *
 * Description :-
 * This function adds a new Term name to the list TermList
 ********************************************************************/
void MV_TermList :: addTerm(char *aTerm)
{
  term_count++;
  if (term_count <= No_of_Terms)
    {
      TermList[term_count] = new char[strlen(aTerm) + 1];
      strcpy(TermList[term_count], aTerm);
    }
  else
    cout << "\n\n *** ERROR : Adding too many items to the list !!";
}


/********************************************************************
 * Function Name : MV_AreaList :: addArea
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area to be added in the list.
 *
 * Outputs :-
 * 1. An entry for the new Area name in the list AreaList
 *
 * Description :-
 * This function adds a new Area name to the list AreaList
 ********************************************************************/
void MV_AreaList :: addArea(char *anArea)
{
  area_count++;
  if (area_count <= No_of_Areas)
    {
      AreaList[area_count] = new char[strlen(anArea) + 1];
      strcpy(AreaList[area_count], anArea);
    }
  else
    cout << "\n\n *** ERROR : Adding too many items to the list !!";
```

```
}


/*************************************************************************
 * Function Name : MV_RelationshipList :: addRelationship
 *
 * Inputs :-
 *1. aRel(Type char *) : The Relationship to be added in the list.
 *
 * Outputs :-
 *1. An entry for the new Relationship name in the list RelList.
 *
 * Description :-
 *    This function adds a new Relationship name to the list RelList.
 *************************************************************************/
void MV_RelationshipList :: addRelationship(char *aRel)
{
  rel_count++;
  if (rel_count <= No_of_Rels)
    {
      RelList[rel_count] = new char[strlen(aRel) + 1];
      strcpy(RelList[rel_count], aRel);
    }
  else
    cout << "\n\n *** ERROR : Adding too many items to the list !!";
}


/*************************************************************************
 * Function Name : MV_AttributeList :: addAttribute
 *
 * Inputs :-
 * 1. anAttr(Type char *) : The Attribute to be added in the list.
 *
 * Outputs :-
 * 1. An entry for the new Attribute name in the list AttrList
 *
 * Description :-
 * This function adds a new Attribute name to the list AttrList.
 *************************************************************************/
void MV_AttributeList :: addAttribute(char *anAttr)
{
  attr_count++;
  if (attr_count <= No_of_Attrs)
```

```
    {
      AttrList[attr_count] = new char[strlen(anAttr) + 1];
      strcpy(AttrList[attr_count], anAttr);
    }
  else
    cout << "\n\n *** ERROR : Adding too many items to the list !!";
}


/*********************************************************************
 * Function Name : MV_ValueList :: addValue
 *
 * Inputs :-
 * 1. aVal(Type char *) : The Value to be added in the list.
 *
 * Outputs :-
 * 1. An entry for the new Value in the list ValList.
 *
 * Description :-
 * This function adds a new Value to the list ValList.
 *********************************************************************/
void MV_ValueList :: addValue(char *aVal)
{
  val_count++;
  if (val_count <= No_of_Vals)
    {
      ValList[val_count] = new char[strlen(aVal) + 1];
      strcpy(ValList[val_count], aVal);
    }
  else
    cout << "\n\n *** ERROR : Adding too many items to the list !!";
}



/*********************************************************************
 * Function Name : MV_TermRelPairs :: addTerm
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term to be added in the list of pairs.
 * 2. nrels(Type int) : The no. of relationships for this new Term.
 *
  * Outputs :-
 * 1. An entry for the new Term name in the list.
```

```
 * 2. An empty list is created to store the relationships.
 *
 * Description :-
 * This function adds a new Term and initializes all the
 * data structures needed to store its relationships.
 ****************************************************************/
void MV_TermRelPairs :: addTerm(char *aTerm, int nrels)
{
  pair_count++;
  Pairs[pair_count].Term = new char[strlen(aTerm) + 1];
  strcpy(Pairs[pair_count].Term, aTerm);

  Pairs[pair_count].No_of_Rels = nrels;
  Pairs[pair_count].Relationships = (char **) new char *[nrels];
  rel_count = -1;
}


/****************************************************************
 * Function Name : MV_TermRelPairs :: addRelationship
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term for which relationship is to be
 *   added.
 * 2. aRel(Type char *) : The relationship to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the relationship name in the list.
 *
 * Description :-
 * This function adds a relationship to the list of
 * relationships for the current Term. You should add all the
 * relationships for a Term one after another (not randomly).
 ****************************************************************/
void MV_TermRelPairs :: addRelationship(char *aTerm, char *aRel)
{
  if ((strcmp(Pairs[pair_count].Term, aTerm)) != 0)
    {
      cout << "\n\n\n Error : You should add all relationships for a
 Term at the same time.";
      return;
    }
  rel_count++;
```

```
  Pairs[pair_count].Relationships[rel_count] =
                      (char *) new char[strlen(aRel) + 1];

  strcpy(Pairs[pair_count].Relationships[rel_count], aRel);
}


/*****************************************************************
 * Function Name : MV_TermRelPairs :: getTermList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. aTermList(Type char **) : The list of terms in the pairs
 * retrieved.
 * 2. no_of_terms(Type int) : Number of terms in the list of
 * pairs.
 *
 * Description :-
 * This function provides a list of all the terms retrieved.
 *****************************************************************/
void MV_TermRelPairs :: getTermList(char **&aTermList,
                                    int &no_of_terms)
{
  aTermList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      aTermList[i] = Pairs[i].Term;
    }
  no_of_terms = No_of_Pairs;
}


/*****************************************************************
 * Function Name : MV_TermRelPairs :: getTermRelationships
 *
 * Inputs :-
 * 1. pos(Type int) : The Term number (amongst all the terms
 * retrieved) for which relationships are requested.
 *
 * Outputs :-
 * 1. aTerm(Type char *) : The Term for which list of
 * relationships is returned.
 * 2. aRelList(Type char **) : A list of all the relationships
```

```
 * for the Term.
 * 3. no_of_rels(Type int) : A count of the number of
 * relationships for that Term.
 *
 * Description :-
 * This function provides a list of all the relationships
 * retrieved for a Term. A value 0 is returned if the data is
 * available and a value 1 is returned if the pos parameter is
 * incorrect.
 ********************************************************************/
int MV_TermRelPairs :: getTermRelationships(int pos, char *&aTerm,
                                char **&aRelList, int &no_of_rels)
{
  if (pos <= No_of_Pairs)
    {
      aTerm = Pairs[pos].Term;     // pos - 1   ????
      aRelList = Pairs[pos].Relationships;
      no_of_rels = Pairs[pos].No_of_Rels;
      return 0;
    }
  else
    return 1;
}


/********************************************************************
 * Function Name : MV_AreaRelPairs :: addArea
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area to be added in the list of
 * pairs.
 * 2. nrels(Type int) : The no. of relationships for this
 * new Area.
 *
 * Outputs :-
 * 1. An entry for the new Area name in the list.
 * 2. An empty list is created to store the relationships.
 *
 * Description :-
 * This function adds a new Area and initializes all the
 * data structures needed to store its relationships.
 ********************************************************************/
void MV_AreaRelPairs :: addArea(char *anArea, int nrels)
{
```

```
  pair_count++;
  Pairs[pair_count].Area = new char[strlen(anArea) + 1];
  strcpy(Pairs[pair_count].Area, anArea);

  Pairs[pair_count].No_of_Rels = nrels;
  Pairs[pair_count].Relationships = (char **) new char *[nrels];
  rel_count = -1;
}


/*********************************************************************
 * Function Name : MV_AreaRelPairs :: addRelationship
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area for which relationship is
 * to be added.
 * 2. aRel(Type char *) : The relationship to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the relationship name in the list.
 *
 * Description :-
 * This function adds a relationship to the list of
 * relationships for the current Area. You should add all the
 * relationships for an Area one after another (not randomly).
 *********************************************************************/
void MV_AreaRelPairs :: addRelationship(char *anArea, char *aRel)
{
  if ((strcmp(Pairs[pair_count].Area, anArea)) != 0)
    {
      cout << "\n\n\n Error : You should add all relationships for a
 Area at the same time.";
      return;
    }
  rel_count++;

  Pairs[pair_count].Relationships[rel_count] =
                      (char *) new char[strlen(aRel) + 1];

  strcpy(Pairs[pair_count].Relationships[rel_count], aRel);
}


/*********************************************************************
```

```
 * Function Name : MV_AreaRelPairs :: getAreaList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. anAreaList(Type char **) : The list of areas in the pairs
 * retrieved.
 * 2. no_of_areas(Type int) : Number of areas in the list of pairs.
 *
 * Description :-
 * This function provides a list of all the areas retrieved.
 ****************************************************************/
void MV_AreaRelPairs :: getAreaList(char **&anAreaList,
                                    int &no_of_areas)
{
  anAreaList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      anAreaList[i] = Pairs[i].Area;
    }
  no_of_areas = No_of_Pairs;
}


/****************************************************************
 * Function Name : MV_AreaRelPairs :: getAreaRelationships
 *
 * Inputs :-
 * 1. pos(Type int) : The Area number (amongst all the Areas
 * retrieved) for which relationships are requested.
 *
 * Outputs :-
 * 1. anArea(Type char *) : The Area for which list of
 * relationships is returned.
 * 2. aRelList(Type char **) : A list of all the relationships
 * for the Area.
 * 3. no_of_rels(Type int) : A count of the number of
 * relationships for that Area.
 *
 * Description :-
 * This function provides a list of all the relationships retrieved
 * for a Area. A value 0 is returned if the data is available and a
 * value 1 is returned if the pos parameter is incorrect.
 ****************************************************************/
```

```
int MV_AreaRelPairs :: getAreaRelationships(int pos, char *&anArea,
                                  char **&aRelList, int &no_of_rels)
{
  if (pos <= No_of_Pairs)
    {
      anArea = Pairs[pos].Area;     // pos - 1   ????
      aRelList = Pairs[pos].Relationships;
      no_of_rels = Pairs[pos].No_of_Rels;
      return 0;
    }
  else
    return 1;
}



/*********************************************************************
 * Function Name : MV_TermAttrPairs :: addTerm
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term to be added in the list of pairs.
 * 2. nattrs(Type int) : The no. of attributes for this new Term.
 *
 * Outputs :-
 * 1. An entry for the new Term name in the list.
 * 2. An empty list is created to store the attributes.
 *
 * Description :-
 * This function adds a new Term and initializes all the
 * data structures needed to store its attributes.
 *********************************************************************/
void MV_TermAttrPairs :: addTerm(char *aTerm, int nattrs)
{
  pair_count++;
  Pairs[pair_count].Term = new char[strlen(aTerm) + 1];
  strcpy(Pairs[pair_count].Term, aTerm);

  Pairs[pair_count].No_of_Attrs = nattrs;
  Pairs[pair_count].Attributes = (char **) new char *[nattrs];
  attr_count = -1;
}


/*********************************************************************
```

```
 * Function Name : MV_TermAttrPairs :: addAttribute
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term for which attribute is to
 * be added.
 * 2. anAttr(Type char *) : The attribute to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the attribute name in the list.
 *
 * Description :-
 * This function adds an attribute to the list of
 * attributes for the current Term. You should add all the
 * attributes for a Term one after another (not randomly).
 ***********************************************************************/
void MV_TermAttrPairs :: addAttribute(char *aTerm, char *anAttr)
{
  if ((strcmp(Pairs[pair_count].Term, aTerm)) != 0)
    {
      cout << "\n\n\n Error : You should add all attributes for a
Term at the same time.";
      return;
    }
  attr_count++;

  Pairs[pair_count].Attributes[attr_count] =
                    (char *) new char[strlen(anAttr) + 1];

  strcpy(Pairs[pair_count].Attributes[attr_count], anAttr);
}

/***********************************************************************
 * Function Name : MV_TermAttrPairs :: getTermList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. aTermList(Type char **) : The list of terms in the pairs
 * retrieved.
 * 2. no_of_terms(Type int) : Number of terms in the list of pairs.
 *
 * Description :-
 * This function provides a list of all the terms retrieved.
 ***********************************************************************/
```

```
void MV_TermAttrPairs :: getTermList(char **&aTermList,
                                     int &no_of_terms)
{
  aTermList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      aTermList[i] = Pairs[i].Term;
    }
  no_of_terms = No_of_Pairs;
}


/*****************************************************************
 * Function Name : MV_TermAttrPairs :: getTermAttributes
 *
 * Inputs :-
 * 1. pos(Type int) : The Term number (amongst all the terms
 * retrieved) for which attributes are requested.
 *
 * Outputs :-
 * 1. aTerm(Type char *) : The Term for which list of attributes
 * is returned.
 * 2. anAttrList(Type char **) : A list of all the attributes
 * for the Term.
 * 3. no_of_attrs(Type int) : A count of the number of attributes
 * for that Term.
 *
 * Description :-
 * This function provides a list of all the attributes retrieved
 * for a Term. A value 0 is returned if the data is available and
 * a value 1 is returned if the pos parameter is incorrect.
 *****************************************************************/
int MV_TermAttrPairs :: getTermAttributes(int pos, char *&aTerm,
                          char **&anAttrList, int &no_of_attrs)
{
  if (pos <= No_of_Pairs)
    {
      aTerm = Pairs[pos].Term;     // pos - 1   ????
      anAttrList = Pairs[pos].Attributes;
      no_of_attrs = Pairs[pos].No_of_Attrs;
      return 0;
    }
  else
    return 1;
```

```
}


/*********************************************************************
 * Function Name : MV_AreaAttrPairs :: addArea
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area to be added in the list of
 * pairs.
 * 2. nattrs(Type int) : The no. of attributes for this new Area.
 *
 * Outputs :-
 * 1. An entry for the new Area name in the list.
 * 2. An empty list is created to store the attributes.
 *
 * Description :-
 * This function adds a new Area and initializes all the
 * data structures needed to store its attributes.
 *********************************************************************/
void MV_AreaAttrPairs :: addArea(char *anArea, int nattrs)
{
  pair_count++;
  Pairs[pair_count].Area = new char[strlen(anArea) + 1];
  strcpy(Pairs[pair_count].Area, anArea);

  Pairs[pair_count].No_of_Attrs = nattrs;
  Pairs[pair_count].Attributes = (char **) new char *[nattrs];
  attr_count = -1;
}


/*********************************************************************
 * Function Name : MV_AreaAttrPairs :: addAttribute
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area for which attribute is to
 * be added.
 * 2. anAttr(Type char *) : The attribute to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the attribute name in the list.
 *
```

```
 * Description :-
 * This function adds an attribute to the list of
 * attributes for the current Area. You should add all the
 * attributes for a Area one after another (not randomly).
 ****************************************************************/
void MV_AreaAttrPairs :: addAttribute(char *anArea, char *anAttr)
{
  if ((strcmp(Pairs[pair_count].Area, anArea)) != 0)
    {
      cout << "\n\n\n Error : You should add all attributes for a
 Area at the same time.";
      return;
    }
  attr_count++;

  Pairs[pair_count].Attributes[attr_count] =
                    (char *) new char[strlen(anAttr) + 1];

  strcpy(Pairs[pair_count].Attributes[attr_count], anAttr);
}


/****************************************************************
 * Function Name : MV_AreaAttrPairs :: getAreaList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. anAreaList(Type char **) : The list of areas in the pairs
 * retrieved.
 * 2. no_of_areas(Type int) : Number of areas in the list of pairs.
 *
 * Description :-
 * This function provides a list of all the areas retrieved.
 ****************************************************************/
void MV_AreaAttrPairs :: getAreaList(char **&anAreaList,
                                     int &no_of_areas)
{
  anAreaList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      anAreaList[i] = Pairs[i].Area;
    }
  no_of_areas = No_of_Pairs;
```

```
}


/*********************************************************************
 * Function Name : MV_AreaAttrPairs :: getAreaAttributes
 *
 * Inputs :-
 * 1. pos(Type int) : The Area number (amongst all the Areas
 * retrieved) for which attributes are requested.
 *
 * Outputs :-
 * 1. anArea(Type char *) : The Area for which list of
 * attributes is returned.
 * 2. anAttrList(Type char **) : A list of all the attributes
 * for the Area.
 * 3. no_of_attrs(Type int) : A count of the number of
 * attributes for that Area.
 *
 * Description :-
 * This function provides a list of all the attributes retrieved
 * for a Area. A value 0 is returned if the data is available and
 * a value 1 is returned if the pos parameter is incorrect.
 *********************************************************************/
int MV_AreaAttrPairs :: getAreaAttributes(int pos, char *&anArea,
                            char **&anAttrList, int &no_of_attrs)
{
  if (pos <= No_of_Pairs)
    {
      anArea = Pairs[pos].Area;     // pos - 1   ????
      anAttrList = Pairs[pos].Attributes;
      no_of_attrs = Pairs[pos].No_of_Attrs;
      return 0;
    }
  else
    return 1;
}



/*********************************************************************
 * Function Name : MV_RelTermPairs :: addRelTermPair
 *
 * Inputs :-
 * 1. aRel(Type char *) : The Relationship to be added in the
```

```
 * list of pairs.
 * 2. aTerm (Type char *) : The Term to be added in the list of
 * pairs.
 *
 * Outputs :-
 * 1. An entry for the new Relationship and Term names in the
 * list.
 *
 * Description :-
 *This function adds a new Relationship and Term to the list of
 * relationship-term pairs.
 ***********************************************************************/
void MV_RelTermPairs :: addRelTermPair(char *aRel, char *aTerm)
{
  pair_count++;
  Pairs[pair_count].Relationship = new char[strlen(aRel) + 1];
  strcpy(Pairs[pair_count].Relationship, aRel);
  Pairs[pair_count].Term = new char[strlen(aTerm) + 1];
  strcpy(Pairs[pair_count].Term, aTerm);
}


/***********************************************************************
 * Function Name : MV_RelTermPairs :: getRelationshipTerm
 *
 * Inputs :-
 * 1. pos(Type int) : The number for the requested pair.
 *
 * Outputs :-
 * 1. aRel(Type char *) : The Relationship in the pair.
 * 2. aTerm (Type char *) : The Term in the pair.
 *
 * Description :-
 * This function provides the relationship and term in a
 * particular pair of the list. A value 0 is returned if the data is
 * available and a value 1 is returned if the pos parameter is
 * incorrect.
 ***********************************************************************/
int MV_RelTermPairs :: getRelationshipTerm(int pos, char *&aRel,
                                           char *&aTerm)
{
  if (pos <= No_of_Pairs)
    {
      aRel = Pairs[pos].Relationship;     // pos - 1 ??
```

```
      aTerm = Pairs[pos].Term;
      return 0;
    }
  else
    return 1;
}


/*********************************************************************
 * Function Name : MV_RelAreaPairs :: addRelAreaPair
 *
 * Inputs :-
 * 1. aRel(Type char *) : The Relationship to be added in the
 * list of pairs.
 * 2. anArea (Type char *) : The Area to be added in the list
 * of pairs.
 *
 * Outputs :-
 * 1. An entry for the new Relationship and Area names in the list.
 *
 * Description :-
 * This function adds a new Relationship and Area to the list of
 * relationship-area pairs.
 *********************************************************************/
void MV_RelAreaPairs :: addRelAreaPair(char *aRel, char *anArea)
{
  pair_count++;
  Pairs[pair_count].Relationship = new char[strlen(aRel) + 1];
  strcpy(Pairs[pair_count].Relationship, aRel);
  Pairs[pair_count].Area = new char[strlen(anArea) + 1];
  strcpy(Pairs[pair_count].Area, anArea);
}


/*********************************************************************
 * Function Name : MV_RelAreaPairs :: getRelationshipArea
 *
 * Inputs :-
 * 1. pos(Type int) : The number for the requested pair.
 *
 * Outputs :-
 * 1. aRel(Type char *) : The Relationship in the pair.
 * 2. anArea (Type char *) : The Area in the pair.
```

```
 *
 * Description :-
 * This function provides the relationship and Area in a particular
 * pair of the list. A value 0 is returned if the data is available
 * and a value 1 is returned if the pos parameter is incorrect.
 ************************************************************************/
int MV_RelAreaPairs :: getRelationshipArea(int pos, char *&aRel,
                                           char *&anArea)
{
  if (pos <= No_of_Pairs)
    {
      aRel = Pairs[pos].Relationship;     // pos - 1 ??
      anArea = Pairs[pos].Area;
      return 0;
    }
  else
    return 1;
}



/************************************************************************
 * Function Name : MV_TermPairs :: addTermPair
 *
 * Inputs :-
 * 1. aSourceTerm(Type char *) : The Term from which the given
 * relationship emanates.
 * 2. aDestTerm(Type char *) : The Term pointed to by the given
 * relationship.
 *
 * Outputs :-
 * 1. An entry for the new Term pair in the list.
 *
 * Description :-
 * This function adds a new Term pair to the list.
 ************************************************************************/
void MV_TermPairs :: addTermPair(char *aSourceTerm, char *aDestTerm)
{
  pair_count++;
  Pairs[pair_count].Source_Term = new char[strlen(aSourceTerm) + 1];
  strcpy(Pairs[pair_count].Source_Term, aSourceTerm);
  Pairs[pair_count].Dest_Term = new char[strlen(aDestTerm) + 1];
  strcpy(Pairs[pair_count].Dest_Term, aDestTerm);
}
```

```
/*********************************************************************
 * Function Name : MV_TermPairs :: getTerms
 *
 * Inputs :-
 * 1. pos(Type int) : The number for the requested pair.
 *
 * Outputs :-
 * 1. aSourceTerm(Type char *) : The source term in the pair.
 * 2. aDestTerm(Type char *) : The destination term in the pair.
 *
 * Description :-
 * This function provides the source and destination terms in a
 * particular pair of the list. A value 0 is returned if the data is
 * available and a value 1 is returned if the pos parameter is
 * incorrect.
 *********************************************************************/
int MV_TermPairs :: getTerms(int pos, char *&aSourceTerm,
                             char *&aDestTerm)
{
  if (pos <= No_of_Pairs)
    {
      aSourceTerm = Pairs[pos].Source_Term;     // pos - 1 ??
      aDestTerm = Pairs[pos].Dest_Term;
      return 0;
    }
  else
    return 1;
}



/*********************************************************************
 * Function Name : MV_AreaPairs :: addAreaPair
 *
 * Inputs :-
 * 1. aSourceArea(Type char *) : The Area from which the given
 * relationship emanates.
 * 2. aDestArea(Type char *) : The Area pointed to by the given
 * relationship.
 *
 * Outputs :-
 * 1. An entry for the new Area pair in the list.
```

```
 *
 * Description :-
 * This function adds a new Area pair to the list.
 ************************************************************************/
void MV_AreaPairs :: addAreaPair(char *aSourceArea, char *aDestArea)
{
  pair_count++;
  Pairs[pair_count].Source_Area = new char[strlen(aSourceArea) + 1];
  strcpy(Pairs[pair_count].Source_Area, aSourceArea);
  Pairs[pair_count].Dest_Area = new char[strlen(aDestArea) + 1];
  strcpy(Pairs[pair_count].Dest_Area, aDestArea);
}


/************************************************************************
 * Function Name : MV_AreaPairs :: getAreas
 *
 * Inputs :-
 * 1. pos(Type int) : The number for the requested pair.
 *
 * Outputs :-
 * 1. aSourceArea(Type char *) : The source Area in the pair.
 * 2. aDestArea(Type char *) : The destination Area in the pair.
 *
 * Description :-
 *This function provides the source and destination Areas in a
 * particular pair of the list. A value 0 is returned if the data is
 * available and a value 1 is returned if the pos parameter is
 * incorrect.
 ************************************************************************/
int MV_AreaPairs :: getAreas(int pos, char *&aSourceArea,
                             char *&aDestArea)
{
  if (pos <= No_of_Pairs)
    {
      aSourceArea = Pairs[pos].Source_Area;     // pos - 1 ??
      aDestArea = Pairs[pos].Dest_Area;
      return 0;
    }
  else
    return 1;
```

```
/*********************************************************************
 * Function Name : MV_AreaTermPairs :: addArea
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area to be added in the list of
 * pairs.
 * 2. nterms(Type int) : The no. of terms for this new Area.
 *
 * Outputs :-
 * 1. An entry for the new Area name in the list.
 * 2. An empty list is created to store the terms.
 *
 * Description :-
 * This function adds a new Area and initializes all the
 * data structures needed to store its terms.
 *********************************************************************/
void MV_AreaTermPairs :: addArea(char *anArea, int nterms)
{
  pair_count++;
  Pairs[pair_count].Area = new char[strlen(anArea) + 1];
  strcpy(Pairs[pair_count].Area, anArea);

  Pairs[pair_count].No_of_Terms = nterms;
  Pairs[pair_count].Terms = (char **) new char *[nterms];
  term_count = -1;
}


/*********************************************************************
 * Function Name : MV_AreaTermPairs :: addTerm
 *
 * Inputs :-
 * 1. anArea(Type char *) : The Area for which term is to be added.
 * 2. aTerm(Type char *) : The term to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the term name in the list.
 *
 * Description :-
 * This function adds an term to the list of
 * terms for the current Area. You should add all the
 * terms for a Area one after another (not randomly).
 *********************************************************************/
```

```
void MV_AreaTermPairs :: addTerm(char *anArea, char *aTerm)
{
  if ((strcmp(Pairs[pair_count].Area, anArea)) != 0)
    {
      cout << "\n\n\n Error : You should add all terms for a Area at
 the same time.";
      return;
    }
  term_count++;

  Pairs[pair_count].Terms[term_count] =
                  (char *) new char[strlen(aTerm) + 1];

  strcpy(Pairs[pair_count].Terms[term_count], aTerm);
}


/*****************************************************************
 * Function Name : MV_AreaTermPairs :: getAreaList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. anAreaList(Type char **) : The list of areas in the pairs
 * retrieved.
 * 2. no_of_areas(Type int) : Number of areas in the list of
 * pairs.
 *
 * Description :-
 * This function provides a list of all the areas retrieved.
 *****************************************************************/
void MV_AreaTermPairs :: getAreaList(char **&anAreaList,
                                     int &no_of_areas)
{
  anAreaList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      anAreaList[i] = Pairs[i].Area;
    }
  no_of_areas = No_of_Pairs;
}


/*****************************************************************
```

```
 * Function Name : MV_AreaTermPairs :: getAreaTerms
 *
 * Inputs :-
 * 1. pos(Type int) : The Area number (amongst all the Areas
 * retrieved) for which terms are requested.
 *
 * Outputs :-
 * 1. anArea(Type char *) : The Area for which list of terms
 * is returned.
 * 2. aTermList(Type char **) : A list of all the terms for the Area.
 * 3. no_of_terms(Type int) : A count of the number of terms for that
 * Area.
 *
 * Description :-
 * This function provides a list of all the terms retrieved
 * for a Area. A value 0 is returned if the data is available and
 * a value 1 is returned if the pos parameter is incorrect.
 ********************************************************************/
int MV_AreaTermPairs :: getAreaTerms(int pos, char *&anArea,
                        char **&aTermList, int &no_of_terms)
{
  if (pos <= No_of_Pairs)
    {
      anArea = Pairs[pos].Area;     // pos - 1   ????
      aTermList = Pairs[pos].Terms;
      no_of_terms = Pairs[pos].No_of_Terms;
      return 0;
    }
  else
    return 1;
}



/********************************************************************
 * Function Name : MV_TermTriples :: addTermTriple
 *
 * Inputs :-
 * 1. aSourceTerm(Type char *) : The Term from which the
 * relationship emanates.
 * 2. aRelationship(Type char *) : The relationship between
 * two terms.
 * 3. aDestTerm(Type char *) : The Term pointed to by the
 * relationship.
```

```
 *
 * Outputs :-
 * 1. An entry for the new Term triple in the list.
 *
 * Description :-
 * This function adds a new Term triple to the list.
 ****************************************************************/
void MV_TermTriples :: addTermTriple(char *aSourceTerm,
                              char *aRelationship, char *aDestTerm)
{
  triple_count++;
  Triples[triple_count].Source_Term =
                                new char[strlen(aSourceTerm) + 1];
  strcpy(Triples[triple_count].Source_Term, aSourceTerm);
  Triples[triple_count].Relationship =
                              new char[strlen(aRelationship) + 1];
  strcpy(Triples[triple_count].Relationship, aRelationship);
  Triples[triple_count].Dest_Term = new char[strlen(aDestTerm) + 1];
  strcpy(Triples[triple_count].Dest_Term, aDestTerm);
}


/****************************************************************
 * Function Name : MV_TermTriples :: getTermTriple
 *
 * Inputs :-
 *       1. pos(Type int) : The number for the requested triple.
 *
 * Outputs :-
 * 1. aSourceTerm(Type char *) : The source term in the triple.
 * 2. aRelationship(Type char *) : The relationship in the triple.
 * 3. aDestTerm(Type char *) : The destination term in the triple.
 *
 * Description :-
 * This function provides the source term, relationship and
 * destination term in a particular triple of the list. A value 0 is
 * returned if the data is available and a value 1 is returned if
 * the pos parameter is incorrect.
 ****************************************************************/
int MV_TermTriples :: getTermTriple(int pos, char *&aSourceTerm,
                         char *&aRelationship, char *&aDestTerm)
{
  if (pos <= No_of_Triples)
    {
```

```
      aSourceTerm = Triples[pos].Source_Term;     // pos - 1 ??
      aRelationship = Triples[pos].Relationship;
      aDestTerm = Triples[pos].Dest_Term;
      return 0;
    }
  else
    return 1;
}
```

```
/*********************************************************************
 * Function Name : MV_AreaTriples :: addAreaTriple
 *
 * Inputs :-
 * 1. aSourceArea(Type char *) : The Area from which the
 * relationship emanates.
 * 2. aRelationship(Type char *) : The relationship between
 * two areas.
 * 3. aDestArea(Type char *) : The Area pointed to by the
 * relationship.
 *
 * Outputs :-
 * 1. An entry for the new Area triple in the list.
 *
 * Description :-
 * This function adds a new Area triple to the list.
 *********************************************************************/
void MV_AreaTriples :: addAreaTriple(char *aSourceArea,
                            char *aRelationship, char *aDestArea)
{
  triple_count++;
  Triples[triple_count].Source_Area =
                               new char[strlen(aSourceArea) + 1];
  strcpy(Triples[triple_count].Source_Area, aSourceArea);
  Triples[triple_count].Relationship =
                              new char[strlen(aRelationship) + 1];
  strcpy(Triples[triple_count].Relationship, aRelationship);
  Triples[triple_count].Dest_Area = new char[strlen(aDestArea) + 1];
  strcpy(Triples[triple_count].Dest_Area, aDestArea);
}
```

```
/*************************************************************************
 * Function Name : MV_AreaTriples :: getAreaTriple
 *
 * Inputs :-
 * 1. pos(Type int) : The number for the requested triple.
 *
 * Outputs :-
 * 1. aSourceArea(Type char *) : The source area in the triple.
 * 2. aRelationship(Type char *) : The relationship in the triple.
 * 3. aDestArea(Type char *) : The destination area in the triple.
 *
 * Description :-
 * This function provides the source area, relationship and
 * destination area in a particular triple of the list. A value 0 is
 * returned if the data is available and a value 1 is returned if
 * the pos parameter is incorrect.
 *************************************************************************/
int MV_AreaTriples :: getAreaTriple(int pos, char *&aSourceArea,
                          char *&aRelationship, char *&aDestArea)
{
  if (pos <= No_of_Triples)
    {
      aSourceArea = Triples[pos].Source_Area;     // pos - 1 ??
      aRelationship = Triples[pos].Relationship;
      aDestArea = Triples[pos].Dest_Area;
      return 0;
    }
  else
    return 1;
}



/*************************************************************************
 * Function Name : MV_AttrValPairs :: addAttribute
 *
 * Inputs :-
 * 1. anAttr(Type char *) : The Attr to be added in the list of
 * pairs.
 * 2. nvals(Type int) : The no. of values for this new Attr.
 *
  * Outputs :-
 * 1. An entry for the new Attribute name in the list.
 * 2. An empty list is created to store the values.
```

```
 *
 * Description :-
 * This function adds a new Attribute and initializes all the
 * data structures needed to store its values.
 ****************************************************************/
void MV_AttrValPairs :: addAttribute(char *anAttr, int nvals)
{
  pair_count++;
  Pairs[pair_count].Attribute = new char[strlen(anAttr) + 1];
  strcpy(Pairs[pair_count].Attribute, anAttr);

  Pairs[pair_count].No_of_Vals = nvals;
  Pairs[pair_count].Values = (char **) new char *[nvals];
  val_count = -1;
}


/****************************************************************
 * Function Name : MV_AttrValPairs :: addValue
 *
 * Inputs :-
 * 1. anAttr(Type char *) : The Attr for which value is to be added.
 * 2. aVal(Type char *) : The value to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the value in the list.
 *
 * Description :-
 * This function adds a value to the list of
 * values for the current Attribute. You should add all the
 * values for a Attribute one after another (not randomly).
 ****************************************************************/
void MV_AttrValPairs :: addValue(char *anAttr, char *aVal)
{
  if ((strcmp(Pairs[pair_count].Attribute, anAttr)) != 0)
    {
      cout << "\n\n\n Error : You should add all values for a Attr
 at the same time.";
      return;
    }
  val_count++;

  Pairs[pair_count].Values[val_count] =
                            (char *) new char[strlen(aVal) + 1];
```

```
  strcpy(Pairs[pair_count].Values[val_count], aVal);
}


/*************************************************************************
 * Function Name : MV_AttrValPairs :: getAttributeList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. anAttrList(Type char **) : The list of attributes in the
 * pairs retrieved.
 * 2. no_of_attrs(Type int) : Number of attributes in the list
 * of pairs.
 *
 * Description :-
 * This function provides a list of all the attributes retrieved.
 *************************************************************************/
void MV_AttrValPairs :: getAttributeList(char **&anAttrList,
                                         int &no_of_attrs)
{
  anAttrList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      anAttrList[i] = Pairs[i].Attribute;
    }
  no_of_attrs = No_of_Pairs;
}


/*************************************************************************
 * Function Name : MV_AttrValPairs :: getAttributeValues
 *
 * Inputs :-
 * 1. pos(Type int) : The Attribute number (amongst all the
 * attrs retrieved) for which values are requested.
 *
 * Outputs :-
 * 1. anAttr(Type char *) : The Attr for which list of values
 * is returned.
 * 2. aValList(Type char **) : A list of all the values for the Attr.
 * 3. no_of_vals(Type int) : A count of the number of values for that
 * Attr.
 *
```

```
 * Description :-
 * This function provides a list of all the values retrieved
 * for a Attribute. A value 0 is returned if the data is available
 * and a value 1 is returned if the pos parameter is incorrect.
 ****************************************************************/
int MV_AttrValPairs :: getAttributeValues(int pos, char *&anAttr,
                                char **&aValList, int &no_of_vals)
{
  if (pos <= No_of_Pairs)
    {
      anAttr = Pairs[pos].Attribute;     // pos - 1   ????
      aValList = Pairs[pos].Values;
      no_of_vals = Pairs[pos].No_of_Vals;
      return 0;
    }
  else
    return 1;
}



/****************************************************************
 * Function Name : MV_TermValPairs :: addTerm
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term to be added in the list of pairs.
 * 2. nvals(Type int) : The no. of values for this new Term.
 *
  * Outputs :-
 * 1. An entry for the new Term name in the list.
 * 2. An empty list is created to store the values.
 *
 * Description :-
 * This function adds a new Term and initializes all the
 * data structures needed to store its values.
 ****************************************************************/
void MV_TermValPairs :: addTerm(char *aTerm, int nvals)
{
  pair_count++;
  Pairs[pair_count].Term = new char[strlen(aTerm) + 1];
  strcpy(Pairs[pair_count].Term, aTerm);

  Pairs[pair_count].No_of_Vals = nvals;
  Pairs[pair_count].Values = (char **) new char *[nvals];
```

```
  val_count = -1;
}


/*********************************************************************
 * Function Name : MV_TermValPairs :: addValue
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term for which value is to be added.
 * 2. aVal(Type char *) : The value to be added to the list.
 *
 * Outputs :-
 * 1. An entry for the value in the list.
 *
 * Description :-
 * This function adds a value to the list of
 * values for the current Term. You should add all the
 * values for a Term one after another (not randomly).
 *********************************************************************/
void MV_TermValPairs :: addValue(char *aTerm, char *aVal)
{
  if ((strcmp(Pairs[pair_count].Term, aTerm)) != 0)
    {
      cout << "\n\n\n Error : You should add all values for a Term
 at the same time.";
      return;
    }
  val_count++;

  Pairs[pair_count].Values[val_count] =
                            (char *) new char[strlen(aVal) + 1];
  strcpy(Pairs[pair_count].Values[val_count], aVal);
}


/*********************************************************************
 * Function Name : MV_TermValPairs :: getTermList
 *
 * Inputs : None.
 *
 * Outputs :-
 * 1. aTermList(Type char **) : The list of terms in the pairs
 *    retrieved.
 * 2. no_of_terms(Type int) : Number of terms in the list of pairs.
```

```
 *
 * Description :-
 * This function provides a list of all the terms retrieved.
 ****************************************************************/
void MV_TermValPairs :: getTermList(char **&aTermList,
                                    int &no_of_terms)
{
  aTermList = (char **) new char *[No_of_Pairs];
  for (int i = 0; i < No_of_Pairs; i++)
    {
      aTermList[i] = Pairs[i].Term;
    }
  no_of_terms = No_of_Pairs;
}


/****************************************************************
 * Function Name : MV_TermValPairs :: getTermValues
 *
 * Inputs :-
 * 1. pos(Type int) : The Term number (amongst all the terms
 * retrieved) for which values are requested.
 *
 * Outputs :-
 * 1. aTerm(Type char *) : The Term for which list of values
 * is returned.
 * 2. aValList(Type char **) : A list of all the values for the Term.
 * 3. no_of_vals(Type int) : A count of the number of values for that
 * Term.
 *
 * Description :-
 * This function provides a list of all the values retrieved
 * for a Term. A value 0 is returned if the data is available and a
 * value 1 is returned if the pos parameter is incorrect.
 ****************************************************************/
int MV_TermValPairs :: getTermValues(int pos, char *&aTerm,
                                     char **&aValList, int &no_of_vals)
{
  if (pos <= No_of_Pairs)
    {
      aTerm = Pairs[pos].Term;     // pos - 1   ????
      aValList = Pairs[pos].Values;
      no_of_vals = Pairs[pos].No_of_Vals;
      return 0;
```

```
    }
  else
    return 1;
}


/*********************************************************************
 * Function Name : MV_TermAttrValTriples :: addTermAttribute
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term in the triple.
 * 2. anAttribute(Type char *) : The attribute in the triple.
 * 3. nvals(Type int) : The number of values for that attribute.
 *
 * Outputs :-
 * 1. An entry for the new Term-Attribute in the list.
 *
 * Description :-
 * This function adds a new Term-attribute pair that is a
 * part of the new triple in the list.
 *********************************************************************/
void MV_TermAttrValTriples :: addTermAttribute(char *aTerm,
                                    char *anAttribute, int nvals)
{
  triple_count++;
  Triples[triple_count].Term = new char[strlen(aTerm) + 1];
  strcpy(Triples[triple_count].Term, aTerm);
  Triples[triple_count].Attribute =
                           new char[strlen(anAttribute) + 1];
  strcpy(Triples[triple_count].Attribute, anAttribute);

  Triples[triple_count].No_of_Vals = nvals;

  Triples[triple_count].Values = (char **) new char *[nvals];
  val_count = -1;
}


/*********************************************************************
 * Function Name : MV_TermAttrValTriples :: addValue
 *
 * Inputs :-
 * 1. aTerm(Type char *) : The Term for which value is being added.
 * 2. anAttribute(Type char *) : The attribute whose value is being
```

```
 * added.
 * 3. aValue(Type char *) : The value being added to the triple.
 *
 * Outputs :-
 * 1. An entry for the new value in the present triple.
 *
 * Description :-
 * This function adds a new value for the current Term-Attribute
 * pair.
 ************************************************************************/
void MV_TermAttrValTriples :: addValue(char *aTerm,
                                       char *anAttribute, char *aValue)
{
  if (((strcmp(Triples[triple_count].Term, aTerm)) != 0) ||
      ((strcmp(Triples[triple_count].Attribute, anAttribute)) != 0))
    {
      cout << "\n\n\n Error : You should add all values for a
 Term-attribute pair at the same time.";
      return;
    }

  val_count++;
  Triples[triple_count].Values[val_count] =
                              new char[strlen(aValue) + 1];
  strcpy(Triples[triple_count].Values[val_count], aValue);
}


/************************************************************************
 * Function Name : MV_TermAttrValTriples :: getTermAttrValTriple
 *
 * Inputs :-
 *       1. pos(Type int) : The number for the requested triple.
 *
 * Outputs :-
 * 1. aTerm(Type char *) : The  term in the triple.
 * 2. anAttribute(Type char *) : The attribute in the triple.
 * 3. aValueList(Type char **) : The list of values in the triple.
 * 4. no_of_vals(Type int) : The number of values in that triple.
 *
 * Description :-
 * This function provides the term, attribute and list of that
 * attribute's values in a particular triple of the list. A value 0
 * is returned if the data is available and a value 1 is returned if
```

```
 * the pos parameter is incorrect.
 ************************************************************************/
int MV_TermAttrValTriples :: getTermAttrValTriple(int pos,
                              char *&aTerm, char *&anAttribute,
                              char **&aValueList, int &no_of_vals)
{
  if (pos <= No_of_Triples)
    {
      aTerm = Triples[pos].Term;     // pos - 1 ??
      anAttribute = Triples[pos].Attribute;
      aValueList = Triples[pos].Values;
      no_of_vals = Triples[pos].No_of_Vals;
      return 0;
    }
  else
    return 1;
}



/************************************************************************
 * Function Name : MV_API_Errors :: HandleError
 *
 * Inputs :-
 *       None.
 *
 * Outputs :-
 * 1. Based on the Error_code, appropriate error message is
 * displayed.
 *
 * Description :-
 * This function utilizes the Error_code data member to provide
 * the appropriate error message(s).
 ************************************************************************/
void MV_API_Errors :: HandleError()
{
  switch(Error_code)
    {
    case NO_ERROR :           // No error
      break;

    case ERR_INPUT :
      cout << "\nERROR : The input(s) to the API function is(are)
 invalid";
```

```
      break;

    case ERR_DATA_ABSENT :
      cout << "\nERROR : The requested data is not available in the
 vocabulary";
      break;

    case ERR_OPEN_FAIL :
      cout << "\nERROR : An attempt to open the Vocabulary has failed
 !!";
      break;

    case ERR_DBNAME_ABSENT :
      cout << "\nERROR : The DBNAME environment variable has not been
 set";
      break;

    default :
      cout << "\nUNKNOWN ERROR !!!";
      break;
    }
}
```

## B.2 Function Implementation

### Example 1

```
/*********************************************************************
 * MV_List_All_Children_of_Term.C : Implements the API function
 *                                  MV_LIst_All_Children_of_Term.
 *
 * Created by : Hemant Kothavade          Creation Date : 6/4/96
 * Last Updated : 6/18/96
 *********************************************************************/


#include <stdlib.h>
#include <iostream.h>
#include "Terms.h"
#include "MV_API_IO.h"


MV_API_Errors ErrObj;
```

```
MV_API_Errors *MV_List_All_Children_of_Term(MV_TermName &Source_Term,
                                            MV_TermList &Child_Terms )
{
  int No_of_Children;

  char *dbname;
  if (dbname = getenv("DBNAME"))
    {
      if (!(OC_open(dbname)))
{
  ErrObj.setErrorCode(ERR_OPEN_FAIL);
  return(&ErrObj);
}
    }
  else
    {
  ErrObj.setErrorCode(ERR_DBNAME_ABSENT);
  return(&ErrObj);
    }

  OC_transactionStart();

  OC_Type* TypeA = (OC_Type *) OC_lookup (STR_ROOT_AREA);
  OC_Property *PropertyA = (OC_Property *) OC_lookup (STR_ID_SEARCH);

  ROOT_AREA *search_term = OC_null;

  OC_InstanceIterator Itr(TypeA,PropertyA,Source_Term.getTermName(),
                          Source_Term.getTermName());
  while(Itr.moreData())
    {
      search_term = (ROOT_AREA*)(OC_Entity *) Itr();
    }

  if (search_term == OC_null)   // Term not found in vocabulary !!
    {
  ErrObj.setErrorCode(ERR_INPUT);
  return(&ErrObj);
    }

  OC_Set *ChildrenSet = search_term -> getSUPERCLASS_OF();
  OC_SetIterator *anIterator =
                (OC_SetIterator *) ChildrenSet -> getIterator();
```

```
  No_of_Children = (int) ChildrenSet -> cardinality();

  Child_Terms.createList(No_of_Children);

  char *tname;
  while(anIterator -> moreData())
    {
      ROOT_AREA *anObject =
            (ROOT_AREA *) (OC_Entity *)(anIterator -> operator()());

      tname = anObject -> getNAME();
      Child_Terms.addTerm(tname);
    }

 // Should we use OC-transactionCommit ?? - NO !!!!
  OC_transactionAbort();
  OC_close();

  return NULL;
}
```

Example 2

```
/*********************************************************************
 * MV_List_All_Area_LocalRelationship_Pairs.C :
 *   Implements the API function
 *                        MV_List_All_Area_LocalRelationship_Pairs
 *
 * Created by : Hemant Kothavade          Creation Date : 8/21/96
 * Last Updated : 8/21/96
 *********************************************************************/


#include <stdlib.h>
#include <iostream.h>
#include <Property.h>
#include "Terms.h"
#include "MV_API_IO.h"
#include "MV_API_Prototypes.h"

MV_API_Errors MyErrObj;

MV_API_Errors *
```

```
MV_List_All_Area_LocalRelationship_Pairs (MV_AreaRelPairs
                                                    &anAreaRelPair)
{
  // First, we get a list of all the areas in the vocabulary. This
  // is easily done by using the already existing
  // MV_List_All_Descendants_of_Area API function.

  MV_AreaName Source_Area;
  MV_AreaList Descendant_Areas;

  Source_Area.setAreaName(STR_ROOT_AREA);

  MV_API_Errors *anError;
  if ((anError = MV_List_All_Descendants_of_Area(Source_Area,
                                          Descendant_Areas)) != NULL)
    {
      MyErrObj = *anError;
      return(&MyErrObj);
    }

  // Now we find out the relationships for each area
  char *dbname;
  if (dbname = getenv("DBNAME"))
    {
      if (!(OC_open(dbname)))
{
  MyErrObj.setErrorCode(ERR_OPEN_FAIL);
  return(&MyErrObj);
}
    }
  else
    {
  MyErrObj.setErrorCode(ERR_DBNAME_ABSENT);
  return(&MyErrObj);
    }

  int no_of_areas = Descendant_Areas.getAreaCount();
  char **Descendants = Descendant_Areas.getAreaList();
// The root area is not in the descendants list, hence the +1
  char **All_areas = (char **) new char *[no_of_areas + 1];

  All_areas[0] = (char *) new char[strlen(STR_ROOT_AREA) + 1];
  strcpy(All_areas[0], STR_ROOT_AREA);
```

```
  for (int i = 1; i <= no_of_areas; i++)
    {
      All_areas[i] = (char *) new char[strlen(Descendants[i - 1])+1];
      strcpy(All_areas[i],Descendants[i - 1]);
    }

  no_of_areas++;

  anAreaRelPair.createList(no_of_areas);

  OC_transactionStart();

  // Get the set in the OOHVR that contains the names of all the
  // relationships
  OC_Set *RelSet=(OC_Set *)OC_lookup(STR_RELATIONSHIP_SET);

  for (i = 0; i < no_of_areas; i++)
    {
      // The relationships for each area are located
      OC_Type* area_type = (OC_Type *) OC_lookup (All_areas[i]);
      OC_PropertyIterator * propertyItr=
                              new OC_PropertyIterator(area_type);
      OC_Property *theProperty = OC_null;;
      bool_type rel_found;
      int no_of_area_rels = 0;

      while (propertyItr->moreData())
{
  theProperty = (OC_Property *)propertyItr->operator()();
  char *prop_name = theProperty->name();
  while (*prop_name != ':')
    prop_name++;
  prop_name += 2;

  rel_found = FALSE;
  // Determine if the property is a relationship
  OC_Iterator *RelIter = RelSet->getIterator();
  while(RelIter->moreData())
    {
      OC_String *aStr =
                  (OC_String *)(OC_Entity *)(RelIter->operator()());
      if ((strcmp(prop_name, aStr->operator char*())) == 0)
{
  rel_found = TRUE;
```

```
  break;
}
    }

  if (rel_found)    // prop_name is a relationship
    no_of_area_rels++;
}

      anAreaRelPair.addArea(All_areas[i], no_of_area_rels);
      OC_PropertyIterator *propertyItr2=
                              new OC_PropertyIterator(area_type);

      while (propertyItr2->moreData())
{
  theProperty = (OC_Property *)propertyItr2->operator()();
  char *prop_name = theProperty->name();
  while (*prop_name != ':')
    prop_name++;
  prop_name += 2;

  rel_found = FALSE;
  // Determine if the property is a relationship
  OC_Iterator *RelIter = RelSet->getIterator();
  while(RelIter->moreData())
    {
      OC_String *aStr =
                 (OC_String *)(OC_Entity *)(RelIter->operator()());
      if ((strcmp(prop_name, aStr->operator char*())) == 0)
{
  rel_found = TRUE;
  break;
}
    }

  if (rel_found)    // prop_name is a relationship
    anAreaRelPair.addRelationship(All_areas[i], prop_name);
}
    }

  OC_transactionAbort();
  OC_close();

  return NULL;
}
```

## REFERENCES


1. AT&T, Inc. Murray Hill, NJ. *UNIX System V Programmer's Guide*, 1990.

2. AT&T, Inc. Murray Hill, NJ. *UNIX System V User's Reference Manual*, 1990.

3. Grady Booch. *Object-Oriented Design.* Benjamin/Cummings Publishing Co., Inc., Redwood City, CA, 1991.

4. Ronald J. Brachman. On the epistemological status of semantic networks. In N. V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 3–50. Academic Press, Inc., New York, NY, 1979.

5. J. Cimino, P. Clayton, G. Hripcsak, and S. Johnson. Knowledge-based approaches to the maintenance of a large controlled medical terminology. *Journal of the American Medical Informatics Association*, 1(1):35–50, 1994.

6. College of American Pathologists, Skokie, IL. *Systematized Nomenclature of Medicine*, second edition, 1982.

7. D. H. Fischer. Consistency rules and triggers for thesauri. *Int. Classif.*, 18(4):212–225, 1991.

8. D. H. Fischer. Consistency rules and triggers for multilingual terminology. In *Proc. TKE'93, Terminology and Knowledge Engineering*, pages 333–342, 1993.

9. C. A. Goble, A. J. Glowinski, W. A. Nolan, and A. L. Rector. A descriptive semantic formalism for medicine. In *Proc. 9th ICDE*, pages 624–631, Vienna, Austria, 1993.

10. H. Gu, J. Cimino, M. Halper, J. Geller, and Y. Perl. Utilizing OODB schema modeling for (medical) vocabulary management. Research Report CIS-96-03, NJIT, 1996. Submitted for conference publication.

11. Michael Halper, James Geller, Yehoshua Perl, and Erich J. Neuhold. A graphical schema representation for object-oriented databases. In R. Cooper, editor, *Interfaces to Database Systems*, pages 282–307. Springer-Verlag, London, 1993.

12. Michael Hammer and Dennis McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, 1981.

13. R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

14. N. Ide, J. Le Maitre, and J. Véronis. Outline of a model for lexical databases. *Information Processing and Management*, 29(2):159–186, 1993.

15. Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In *Proc. 1992 ACM SIGMOD Conference on Management of Data*, San Diego, CA, June 1992.

16. Won Kim and Frederick H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Press, New York, NY, 1989.

17. D. Kumar and S. C. Shapiro. Architecture of an intelligent agent in SNePS. *SIGART Special Section on Integrated Cognitive Architectures*, 2(4), 1991.

18. D. Kumar and S. C. Shapiro. Modeling a rational cognitive agent in SNePS. In P. Barahona, L. Moniz Pereira, and A. Porto, editors, *EPIA 91: 5th Portugese Conference on Artificial Intelligence, Lecture Notes in Artificial Intelligence 541*, pages 120–134. Springer-Verlag, Heidelberg, Germany, 1991.

19. Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1990.

20. L. Liu, H. Gu, J. Geller, and Y. Perl. Modeling a vocabulary in an object-oriented database. Research Report CIS-96-11, NJIT, 1996. Submitted for conference publication.

21. G. A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

22. W. Möhr and L. Rostek. TEDI: An object-oriented terminology editor. In *Proc. TKE'93, Terminology and Knowledge Engineering*, pages 363–374, 1993.

23. National Library of Medicine, Bethesda, MD. *Medical Subject Headings.* Updated annually.

24. ONTOS, Inc. Lowell, MA. *ONTOS DB 3.1 Reference Manual*, 1995.

25. Valery Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, and $O_2$. *SIGMOD Record*, 21(1):93-104, March 1992.

26. J. F. Sowa. *Principles of Semantic Networks, Explorations in the Representation of Knowledge.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

27. United States National Center for Health Statistics, Washington, DC. *International Classification of Diseases: Ninth Revision, with Clinical Modifications*, 1980.

28. William A. Woods. What's in a link: Foundations for semantic networks. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 218–241. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1985.

29. Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.